ARM® Architecture Reference Manual

Performance Monitors v2 Supplement

ARM®

ARM DDI 0457A (ID122209)

ARM Architecture Reference Manual
Performance Monitors v2 Supplement

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Confidentiality	Change
22 December 2009	A	Non-Confidential	First release. Specification of Performance Monitor v2., and changes from v1.

This document describes the architecture changes between Performance Monitor v1 and Performance Monitor v2. It will be superseded by Issue C of the *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition.*

Proprietary Notice

This document is Non-Confidential but any disclosure by you is subject to you providing notice to and the acceptance by the recipient of, the conditions set out above.

In this document, where the term ARM is used to refer to the company it means "ARM or any of its subsidiaries as appropriate".

Note

The term ARM is also used to refer to versions of the ARM architecture, for example ARMv7 refers to version 7 of the ARM architecture. The context makes it clear when the term is used in this way.

Contents
ARM Architecture Reference Manual Performance Monitors v2 Supplement

Preface

This preface summarizes the contents of this book and lists the conventions it uses. It contains the following sections:

About this book

This book describes the changes to the performance monitors, defined in the ARM®v7 architecture, made in version 2 of the Performance Monitors specification, PMUv2. It is, therefore, a supplement to Issue B of the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

Note

This supplement provides information that ARM will include in issue C of the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*. When ARM publishes that issue it will withdraw this supplement.

Using this book

This book is organized into the following chapters:

Chapter 1 ***Introduction***

Introduces the performance monitors, their status in the ARM architecture, and the changes introduced in version 2 of the performance monitors specification.

Chapter 2 ***Performance Monitors***

Describes version 2 of the performance monitors.

Appendix A ***Recommended Memory-mapped and External Debug Interface for the Performance Monitors***

Describes the optional memory-mapped interface to the performance monitors.

Note

This description is not part of the ARM architecture specification. It is included here as supplementary information, for the convenience of developers and users who might require this information.

Glossary Contains definitions of some terms used in this manual. See the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* for a more complete glossary of terms used in ARM architecture descriptions.

Conventions

The following sections describe conventions that this book can use:

- *General typographic conventions*
- *Signals*
- *Numbers*.

General typographic conventions

`monospaced`	Used for assembler syntax descriptions, pseudocode, and source code examples. Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.
italic	Highlights important notes, introduces special terminology, and denotes internal cross-references and citations.
bold	Denotes signal names and is used for terms in descriptive lists, where appropriate.
SMALL CAPITALS	Used for terms that have specific technical meanings, that are defined in the Glossary.

Signals

In general the Performance Monitors v2 specification does not define processor signals, but it does include some signal examples and recommendations. It uses the following signal conventions:

Signal level	The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means: • HIGH for active-HIGH signals • LOW for active-LOW signals.
Lower-case n	At the start or end of a signal name denotes an active-LOW signal.

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by 0b, and hexadecimal numbers by `0x` and written in a `monospaced` font.

Additional reading

This section lists publications by ARM and by third parties.

See Infocenter, `http://infocenter.arm.com`, for access to ARM documentation.

ARM publications

- *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406).
- *ARM Debug Interface v5 Architecture Specification* (ARM IHI 0031).
- *CoreSight Architecture Specification* (ARM IHI 0029).
- *Embedded Trace Macrocell Architecture Specification* (ARM IHI 0014).
- *CoreSight Program Flow Trace Architecture Specification* (ARM IHI 0035).

Other publications

The following books are referred to in this book, or provide more information:

- IEEE Std 1149.1-2001, *IEEE Standard Test Access Port and Boundary Scan Architecture (JTAG)*.
- JEP106, *Standard Manufacturers Identification Code*, JEDEC Solid State Technology Association.

Feedback

ARM welcomes feedback on its documentation.

Feedback on this book

If you have comments on the content of this book, send e-mail to errata@arm.com. Give:

- the title
- the number, ARM DDI 0457A
- the page numbers to which your comments apply
- a concise explanation of your comments.

Chapter 1
Introduction

This chapter introduces the Performance Monitors, and the changes made in Performance Monitors v2, including the mechanism for identifying the implemented Performance Monitors version. It contains the following section:

- *About the Performance Monitors* on page 1-14.

1.1 About the Performance Monitors

The Performance Monitors are part of the ARM debug architecture. Many ARMv6 processors included performance monitors, but before ARMv7 they were not part of the architecture. Publication of the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* was the first architectural specification of the Performance Monitors, and that specification was derived from the earlier ARM implementations. That specification is now described as Performance Monitors v1.

In ARMv7, the performance monitors are an optional feature of an implementation, but ARM strongly recommends that ARMv7-A and ARMv7-R implementations include the performance monitors.

The basic form of the performance monitors is:

- A cycle counter, with the ability to count every cycle or every sixty-fourth cycle.
- A number of event counters. The event counted by each counter is programmable. ARMv7 provides space for up to 31 counters. The actual number of counters is IMPLEMENTATION DEFINED, and the specification includes an identification mechanism.
- Controls for:
  - enabling and resetting counters
  - flagging overflows
  - enabling interrupts on overflow.

  Monitoring software can enable the cycle counter independently of the event counters.

The original performance monitors specification divides the set of events that can be monitored into:

- events that are likely to be consistent across many microarchitectures
- other events, that are likely to be implementation-specific.

As a result, it defines a common set of events to be used across many microarchitectures, and a large space reserved for IMPLEMENTATION DEFINED events.

The full set of events for any given implementation is IMPLEMENTATION DEFINED. The Performance Monitors v1 specification does not require an implementation to include any of the common set of events, but implementations must not use the numbers allocated for the common set of events except as defined.

ARMv7 defined only a system control coprocessor (CP15) interface to the performance monitor registers.

1.1.1 About the Performance Monitors v2

ARM has enhanced the performance monitors specification, as Performance Monitors v2. Chapter 2 *Performance Monitors* is the Performance Monitors v2 specification. The main changes in Performance Monitors v2 are:

- Filtering of event counting by processor state.

- Changes to the defined set of performance monitor events. In particular, events are now divided into the following groups:
  - common architectural events
  - common microarchitectural events
  - IMPLEMENTATION DEFINED events.
- Performance monitor implementations must implement at least a limited subset of the common events.

Also, ARM now defines an optional memory-mapped interface to the performance monitors, described in Appendix A *Recommended Memory-mapped and External Debug Interface for the Performance Monitors*.

Note

The memory-mapped interface to the performance monitors aligns the performance monitors with other ARM CoreSight debug and trace components.

1.1.2 Identification of the performance monitors version

The introduction of Performance Monitors v2 adds an extra field to the CP15 Debug Feature Register 0, ID_DFR0, that identifies the Performance Monitors version. Figure 1-1 shows the updated ID_DFR0 format.

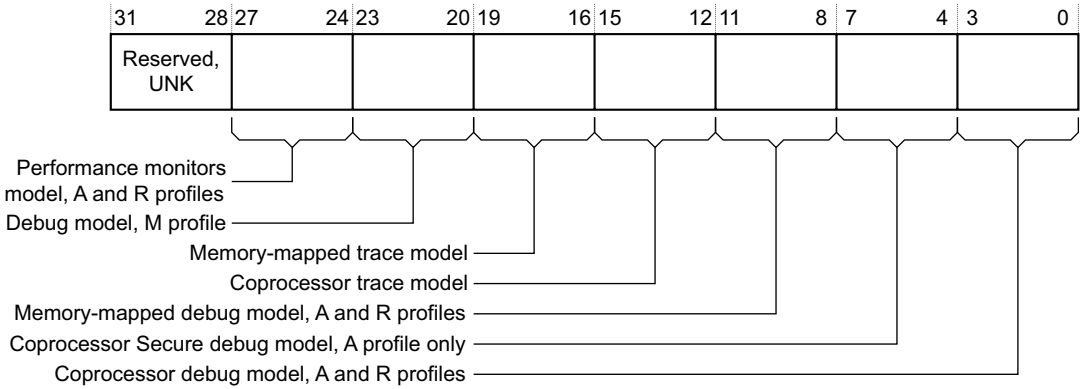


Figure 1-1 ID_DFR0 bit assignments, with Performance Monitors v2 support

The bit assignments of the modified fields in ID_DFR0 are:

Bits [31:28] Reserved, UNK.

Performance monitors model, A and R profiles, bits [27:24]

Support for coprocessor-based performance monitors, for A and R profile processors. Permitted values are:

0b0000 Performance Monitors v2 not supported.

0b0001 Support for Performance Monitors v1.

0b0010 Support for Performance Monitors v2.

0b1111 No performance monitor support.

Note

A value of 0b0000 gives no indication of whether Performance Monitors v1 are supported.

For the definition of the other fields of ID_DFR0, and the register characteristics, see the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

Chapter 2
Performance Monitors

This chapter describes the performance monitors, that are an optional non-invasive debug component. It describes version 2 of the *Performance Monitor Unit* (PMU) architecture, PMUv2, and contains the following sections:

2.1 About the performance monitors

Performance monitors are essential in high-performance, multi-processor SoCs. Complex, multi-processor, SoCs and computer systems are increasingly difficult to model. Part of that difficulty is that these systems comprise multiple interacting and reacting components.

The two main areas of improvement in Performance Monitors v2 are:

- better directed profiling by an operating system software monitor.
- more uniformity between implementations. by extending the list of standard event types to include more common, useful events.

To enable interaction with external monitoring, an implementation might consider additional enhancements, including providing memory-mapped and external debug access to the performance monitor registers. This means the counter resources can be used for system monitoring in a system where they are not used by the software running on the processor. For more information see Appendix A *Recommended Memory-mapped and External Debug Interface for the Performance Monitors*.

The events that might be monitored comprise:

- events likely to be consistent across many microarchitectures. PMUv2 further divides these into architectural and microarchitectural events.
- IMPLEMENTATION DEFINED events.

The PMU architecture identifies events by event numbers and:

- defines event numbers for common events, for use across many architectures and microarchitectures
- reserves a large event number space for IMPLEMENTATION DEFINED events.

When a processor supports monitoring of an event that is assigned a common event number, ARM strongly recommends that it uses that number for the event. However, software might encounter implementations where an event assigned a number in this range is monitored using an event number from the IMPLEMENTATION DEFINED range.

Note

PMUv2 defines additional common event numbers, and future revisions of the PMU architecture might define other common event numbers. This is one reason why software must not assume that an event with an assigned common event number is never monitored using an event number from the IMPLEMENTATION DEFINED range.

2.1.1 PMU architecture versions, and status in the ARM architecture

This chapter describes the PMUv2 architecture version.

2.2 Effect of the Security Extensions

This section describes the effects of the Security Extensions on the performance monitors. It contains the following subsection:

- *Interaction with Security Extensions*.

2.2.1 Interaction with Security Extensions

Effects of non-invasive debug authentication on the performance monitors describes the behavior of the performance monitors when any of the following applies:

- non-invasive debug is disabled
- the processor is in a mode or state where non-invasive debug is not permitted
- the processor is in Debug state.

The non-invasive debug authentication signals do not control the cycle counter, PMCCNTR. However, setting the PMCR.DP bit to 1 disables PMCCNTR counting in regions of code where the event counters are disabled. For more information see *CP15 c9, Performance Monitor Control Register, PMCR* on page 2-40.

Effects of non-invasive debug authentication on the performance monitors

Table 2-1 describes the behavior of the performance monitors when non-invasive debug is disabled or not permitted, and in Debug state.

Table 2-1 Behavior of performance monitors when non-invasive debug not permitted

Debug state	Non-invasive debug permitted and enabled	PMCR.DP[a]	Event counters enabled and events exported[a, b]	PMCCNTR enabled
Yes	x	x	No	No
No	Yes	x	Yes	Yes
	No	0	No	Yes
		1	No	No

a. See *CP15 c9, Performance Monitor Control Register, PMCR* on page 2-40.
b. The events are exported only if the PMCR.X bit is set to 1, see *CP15 c9, Performance Monitor Control Register, PMCR* on page 2-40.

The performance monitors are not intended to be completely accurate, see the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* for more information.

Entry to and exit from Debug state can also disturb the normal running of the processor, causing additional inaccuracy in the performance monitors. Disabling the counters while in Debug state limits the extent of this inaccuracy. Implementations can limit this inaccuracy to a greater extent, for example by disabling the counters as soon as possible during the Debug state entry sequence.

2.3 Event filtering, PMUv2

PMUv2 can filter events by combinations of:

- mode, for example User or privileged modes
- security state, Secure or Non-secure.

This gives software more flexibility for counting events across multiple processes.

An operating system can configure the filtering of the counters. The filtering applies whether the processor is running in:

- Secure state. This is the legacy PMU operation.
- Non-secure state, with a Secure monitor.

An operating system can enable User mode control of event filtering.

Normally, deasserting **SPNIDEN** disables event counting in Secure state. Table 2-2 shows typical configurations an operating system might use.

Table 2-2 Default configurations

	Secure state		Non-secure state	
Modes in which events are counted	**Privileged**	**User**	**Privileged**	**User**
All modes	Yes	Yes	Yes	Yes
Privileged modes	Yes	-	Yes	-
User mode	-	Yes	-	Yes

Note

The encodings for these combinations of mode are backwards compatible, so that if no filter is specified, the default operation is to count in all modes.

2.3.1 Accuracy of event filtering

The PMU architecture does not require event filtering to be accurate. Normally, it is acceptable for an event to leak through from one state to another.

For most events, it is acceptable that, during a transition between states, events generated by instructions executed in one state are counted in the other state. This happens because events are filtered by a perception of the current state generated elsewhere in the processor. Such errors in counting are small compared to the counts when there is no ambiguity and all parts of the processor have the same perception of the execution state. The following sections describe the cases where event counts must not leak into the wrong state:

- *Exception-related events* on page 2-21
- *Software increment events* on page 2-21.

Exception-related events

The PMU must filter events related to exceptions and exception handling according to the mode in which the exception occurred. These events are:

- exception taken
- exception return instruction architecturally executed, condition check pass
- write to CONTEXIDR instruction architecturally executed, condition check pass
- write to translation table base instruction architecturally executed, condition check pass.

The PMU must not count an exception after it has been taken because this could systematically report a result of zero exceptions in User mode. Similarly, it is not acceptable for the PMU to count exception returns or writes to CONTEXTIDR after the return from the exception.

Note

Unprivileged software cannot write to CONTEXTIDR.

Software increment events

The PMU must filter software increment events according to the mode in which the software increment occurred. Software increment counting must also be precise, meaning the PMU must count every architecturally-executed software increment event, and must not count any speculatively-executed software increment.

2.4 Counter enables

The PMCR.E bit is a global counter enable bit, and PMCNTENSET provides an enable bit for each counter, as Table 2-3 shows.

Table 2-3 Event counter enables

PMCR.E	PMCNTENSET[x] is 0	PMCNTENSET[x] is 1
0	PMN*x* disabled	PMN*x* disabled
1	PMN*x* disabled	PMN*x* enabled

For more information about the enable bits, see *CP15 c9, Performance Monitor Control Register, PMCR* on page 2-40, and *CP15 c9, Count Enable Set Register, PMCNTENSET* on page 2-44.

Table 2-4 shows the PMCCNT enables.

Table 2-4 Cycle counter enables

PMCR.E	PMCNTENSET[31] is 0	PMCNTENSET[31] is 1
0	PMCCNT disabled	PMCCNT disabled
1	PMCCNT disabled	PMCCNT enabled

2.4.1 Counter access

Counters, including the cycle counter, are accessible in privileged modes. However, access to the counter registers is subject to the access permissions described in *Access permissions, PMUv2* on page 2-39. In particular, accesses from User mode might be UNDEFINED.

2.5 Event numbers and mnemonics

The following sections describe the event numbers, and the mnemonics for the events:

- *Definition of terms*
- *Event classification differences between PMUv1 and PMUv2* on page 2-26
- *Common architectural event numbers* on page 2-27
- *Common microarchitectural feature event numbers* on page 2-30
- *Implementation defined feature event numbers* on page 2-35.

2.5.1 Definition of terms

Speculatively executed

Many events relate to speculatively executed instructions. Here, speculatively executed means that the processor did some work associated with the instruction, but the instruction was not necessarily architecturally executed.

Every speculatively executed instruction must have been fetched by the processor, and every instruction that is architecturally executed and modified the state of the processor was speculatively executed. An instruction may be split into one or more micro-operations and so an architecturally executed instruction may count as more than one speculatively executed instructions. An instruction that explicitly makes no change the state of the processor, such as a NOP, may not be counted as speculatively executed even though it has been architecturally executed; however, a conditional instruction that fails its condition code and so implicitly makes no change to the state of the processor will also be speculatively executed.

However, it is not required that every instruction that is fetched is counted as one that was speculatively executed.

Different groups of events are permitted to have different implementation defined definitions of speculatively executed. It is also implementation defined in each case whether conditional instructions that fail their condition code check are counted as speculatively executed instructions.

Instruction memory access

A processor acquires instructions for execution through instruction fetches. Instruction fetches may be due to:

- fetching instructions that are architecturally executed
- the result of the execution of an instruction preload instruction (PLI)
- speculation that a particular instruction may be executed in the future.

The relationship between the fetch of an individual instruction and an instruction memory access is IMPLEMENTATION DEFINED. For example, an implementation may fetch many instructions including a non-integer number of instructions in a single instruction memory access.

Memory-read operations

A processor accesses memory through memory-read and memory-write operations. A memory-read operation may be due to:

- the result of an architecturally executed memory-reading instructions
- the result of a speculatively executed memory-reading instructions
- a page table walk.

For levels of cache hierarchy beyond the Level 1 caches, memory-read operations also include accesses made as part of a refill of another cache closer to the processor. Such refills may be due to:

- Memory-read operations or memory-write operations that miss in the cache
- the execution of a data preload instruction (PLD, and, implementation defined, PLDW)
- or a unified cache, the execution of an instruction preload instruction (PLI)
- the execution of a cache maintenance operation

  Note

  A preload instruction or cache maintenance operation is not, in itself, an access to that cache. However, it may generate cache refills which are in turn treated as memory-read operations beyond that cache.

- speculation that a future instruction may access the memory location.

This list is not exhaustive.

The relationship between Memory-reading instructions and Memory-read operations is `IMPLEMENTATION DEFINED`. For example, for some implementations an LDM instruction that reads two registers may generate one memory-read operation if the address is doubleword aligned, whereas for others it generates two memory-read operations.

Memory-write operations

Memory-write operations may be due to:

- the result of an architecturally executed Memory-writing instructions
- the result of a speculatively executed Memory-writing instructions

Note

Speculatively executed memory-writing instructions that do not become architecturally executed must not alter the architecturally defined view of memory. They can, however, generate a Memory-write operation that is later undone in some implementation specific way.

For levels of cache hierarchy beyond the Level 1 caches, Memory-write operations also include accesses made as part of a write-back from another cache closer to the processor. Such write-backs may be due to:

- evicting a dirty line from the cache, to allocate a cache line for a cache refill; see Memory-read operations

- the execution of a cache maintenance operation

  Note

  A cache maintenance operation is not in itself an access to that cache. However, it may generate write-backs which are in turn treated as Memory-write operations beyond that cache.

- the result of a coherency request from another processor.

This list is not exhaustive.

The relationship between Memory-writing instructions and Memory-write operations is IMPLEMENTATION DEFINED. For example, for some implementations an STM instruction that writes two registers may generate one Memory-write operation if the address is doubleword aligned, whereas for others it generates two Memory-write operations. In other implementations, the result of two STR instructions that write to adjacent memory may be merged into a single Memory-write operation.

Note

The data written back from a cache that is shared with other processors may not be data that was written by the processor that performs the operation that leads to the write-back. Nevertheless, the event is counted as a write-back event for that processor.

Instruction architecturally executed

Instruction architecturally executed is a class of event that counts for each instruction of the specified type. Architecturally executed means that the program flow is such that the counted instruction would be executed in an implementation of the simple sequential execution model.

An instruction is architecturally executed if the behavior of the program on the processor is consistent with the instruction having been executed on a simple execution model of the architecture. Therefore an instruction that has been executed and retired is defined to be architecturally executed. In processors that perform speculative execution, an instruction is not architecturally executed if the results of the speculative execution are discarded.

Each architecturally executed instruction is counted once, even if the implementation splits the instruction into multiple operations.

Instructions that have no visible effect on the architectural state of the processor are architecturally executed if they form part of the architecturally executed program flow. The point where such instructions are retired is `IMPLEMENTATION DEFINED`.

Examples of instructions that have no visible effect are:

- NOP
- a conditional instruction that fails its condition code check
- a compare with (not) zero and branch instruction that does not branch.

The point at which an event causes an event counter to be updated is not defined.

Unless otherwise stated, all instructions of the specified type are counted even if they have no visible effect on the architectural state of the processor. This includes a conditional instruction that fails its condition code check.

For events that count only the execution of instructions that update context state, such as writes to the CONTEXTIDR, if such an instruction is executed twice without an intervening ISB, exception entry or exception return, it is UNPREDICTABLE whether

Instruction architecturally executed (condition check pass)

Instruction architecturally executed (condition check pass) is a class of event that explicitly does not occur for:

- a conditional instruction that fails its condition code check
- a compare with (not) zero and branch instruction that does not branch
- a Store-Exclusive instruction that does not write to memory.

Otherwise, the definition of architecturally executed is the same as for *Instruction architecturally executed*.

2.5.2 Event classification differences between PMUv1 and PMUv2

The differences between PMUv1 and PMUv2 are as follows:

PMUv1 Defines only a single list of event numbers covering both architectural and microarchitectural events.

PMUv2

- Splits the list of event numbers into separate architectural and microarchitectural lists.
- Changes the names of some of the events defined in PMUv1. These name changes do not affect what the event counts.
- Defines some additional event numbers.
- Gives some recommendations about assigning event numbers in the IMPLEMENTATION DEFINED event number range.

Table 2-5 shows the organization of the PMU architectural and microarchitectural event numbers in event number order.

Table 2-5 PMU event numbers

Event number	Event category
0x00	*Common architectural event numbers* on page 2-27
0x01 - 0x05	*Common microarchitectural feature event numbers* on page 2-30
0x06 - 0x0F	*Common architectural event numbers* on page 2-27

Table 2-5 PMU event numbers (continued)

Event number	Event category
0x10 - 0x1B	*Common microarchitectural feature event numbers* on page 2-30
0x1C	*Common architectural event numbers*
0x1D	*Common microarchitectural feature event numbers* on page 2-30

2.5.3 Common architectural event numbers

This section describes the PMUv2 list of defined event numbers. See *Event classification differences between PMUv1 and PMUv2* on page 2-26 for information about the PMUv1 classification of event numbers.

For the common features, normally the counters must increment only once for each event. The event descriptions include any exceptions to this rule.

In these definitions, the term architecturally executed means that the instruction flow is such that the counted instruction would have been executed in a simple sequential execution of the program.

Note

Unless otherwise stated, all instructions of the specified type are counted even if they have no visible effect on the architectural state of the processor. This includes a conditional instruction that fails its condition code check.

The common architectural feature event numbers are:

0x00, Instruction architecturally executed, condition check pass, Software increment

The counter increments on writes to the PMSWINC register. For more information see *CP15 c9, Software Increment Register, PMSWINC* on page 2-48.

If the processor performs two architecturally executed writes to the PMSWINC without an intervening ISB, exception entry or exception return, then the event is counted twice.

The PMU architecture mnemonic for this event is PMNC_SW_INCR.

0x06, Instruction architecturally executed, condition check pass, Load

The counter increments for every executed instruction that explicitly reads data, including SWP.

The PMU architecture mnemonic for this event is LD_RETIRED.

0x07, Instruction architecturally executed, condition check pass, Store

The counter increments for every executed instruction that explicitly writes data, including SWP.

The counter does not increment for a Store-Exclusive instruction that fails.

The PMU architecture mnemonic for this event is ST_RETIRED.

0x08, Instruction architecturally executed

The counter increments for every architecturally executed instruction.

The PMU architecture mnemonic for this event is INST_RETIRED.

0x09, Exception taken

The counter increments for each exception taken. See *Exception-related events* on page 2-21.

The PMU architecture mnemonic for this event is EXC_TAKEN.

Note

The counter counts the processor exceptions described in the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*. It does not count floating-point exceptions when floating-point exception traps are disabled or ThumbEE null and index checks.

0x0A, Instruction architecturally executed, condition check pass, Exception return

The counter increments for each executed exception return instruction. The *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* defines the counted instructions. See *Exception-related events* on page 2-21.

The PMU architecture mnemonic for this event is EXC_RETURN.

0x0B, Instruction architecturally executed, condition check pass, Write to CONTEXTIDR

The counter increments for every write to the CONTEXTIDR. See *Exception-related events* on page 2-21.

If the processor performs two architecturally-executed writes to the CONTEXIDR without an intervening ISB, exception entry or exception return, it is UNPREDICTABLE whether the first write to the CONTEXIDR is counted because it is UNPREDICTABLE whether the write has any effect.

The PMU architecture mnemonic for this event is CID_WRITE_RETIRED.

0x0C, Instruction architecturally executed, condition check pass, Software change of the PC

The counter increments for every software change of the PC. This includes all:

- branch instructions
- memory-reading instructions that explicitly write to the PC
- data processing instructions that explicitly write to the PC
- exception generating instructions.

This does not include exceptions, other than exceptions due to exception generating instructions.

It is IMPLEMENTATION DEFINED whether ISB is counted as a software change to the PC or an integer data processing instruction.

The PMU architecture mnemonic for this event is PC_WRITE_RETIRED.

0x0D, Instruction architecturally executed – Immediate branch

The counter increments each time the processor executes one of the following instructions:

- `B{L} <label>`
- `BLX <label>`
- `CB{N}Z <Rn>,<label>`
- In ThumbEE state only, `HB{L} #HandlerId`
- In ThumbEE state only, `HB{L}P #<imm>, #HandlerId`.

The counter counts all immediate branch instructions that are architecturally executed.

The PMU architecture mnemonic for this event is BR_IMMED_RETIRED.

0x0E, Instruction architecturally executed, condition check pass, Procedure return

The counter counts the following procedure return instructions:

- `BX R14`
- `MOV PC,LR`
- `POP {…,PC}`
- `LDR PC,[SP],#offset`
- In ThumbEE state only, `LDMIA R9!,{…,PC}`
- In ThumbEE state only, `LDR PC,[R9],#offset`.

The PMU architecture mnemonic for this event is BR_RETURN_RETIRED.

Note

The counter counts only the listed instructions as procedure returns. For example, it does not count the following as procedure return instructions:

- `BX R0`, because Rm != R14
- `MOV PC,R0`, because Rm != R14
- `LDM SP,{…,PC}`, because writeback is not specified
- `LDR PC,[SP,#offset]`, because this specifies the wrong addressing mode.

0x0F, Instruction architecturally executed, condition check pass, Unaligned load or store

The counter counts each instruction that accesses an unaligned address. That is, the instruction either triggered an alignment fault, or would have done so if the SCTLR.A bit had been 1.

The PMU architecture mnemonic for this event is UNALIGNED_LDST_RETIRED.

See the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* for more information.

0x1C, Instruction architecturally executed, condition check pass, Write to translation table base

The counter counts every write to a translation table base register, TTBR0 or TTBR1. See *Exception-related events* on page 2-21. If the processor executes two writes to a translation table base register without an intervening ISB, exception entry or exception return, it is UNPREDICTABLE whether the first write to translation table base is counted because it is UNPREDICTABLE whether the write has any effect.

The PMU architecture mnemonic for this event is TTBR_WRITE_RETIRED.

2.5.4 Common microarchitectural feature event numbers

This section describes the PMUv2 list of defined event numbers. See *Event classification differences between PMUv1 and PMUv2* on page 2-26 for information about the PMUv1 classification of event numbers.

The common microarchitectural features are features that are likely to be implemented across a wide range of implementations. Unlike the common architectural feature events, there can be some IMPLEMENTATION DEFINED variation between definitions on different implementations.

Unless otherwise stated, the common microarchitectural features relate only to events resulting from the operation of the processor counting the events. Events resulting from the operation of other processors that might share a resource must not be counted. Where a resource can be subject to events that do not result from the operation of any of the ARM processors that share it, ARM recommends that the resource implements its own event counters. An example of a resource that might require its own event counters is a shared Level 2 cache that is subject to accesses from a system coherency port on that cache.

The event definitions relating to Level 2 caches generally assume the Level 2 cache is shared. The event definitions relating to Level 1 caches generally assume the Level 1 cache is not shared.

The common microarchitectural feature event numbers are:

0x01, Level 1 instruction cache refill

The counter counts instruction memory accesses that cause a TLB refill of at least the Level 1 instruction TLB. This includes each instruction memory access that causes an access to a level of memory system due to a translation table walk or an access to another level of TLB caching.

CP15 cache maintenance operations do not count as events.

The PMU architecture mnemonic for this event is L1I_CACHE_REFILL.

0x02, Level 1 instruction TLB refill

The counter counts each instruction memory access that causes an access to a level of the memory system due to a translation table walk or an access to another level of TLB caching.

CP15 TLB maintenance operations do not count as events.

The PMU architecture mnemonic for this event is L1I_TLB_MISS.

0x03, Level 1 data cache refill

The counter counts each memory-read operation or memory-write operation that causes a refill of at least the Level 1 data or unified cache. Accesses that do not cause a new cache refill, but are satisfied from refilling data of a previous miss, are not counted. Each access to a cache line that causes a new linefill is counted, including the multiple accesses of load or store multiples. Accesses to a cache line that generate a memory access but not a new linefill, such as Write-through writes that hit in the cache, are not counted.

Note

The load and store multiple instructions include PUSH and POP.

CP15 cache maintenance operations do not count as events.

The PMU architecture mnemonic for this event is L1D_CACHE_REFILL.

0x04, Level 1 data cache access

The counter counts each memory-read operation or memory-write operation that causes a cache access to at least the Level 1 data or unified cache. Each access to a cache line is counted including the multiple accesses of instructions such as LDM or STM. Each access to other Level 1 data or unified memory structures, for example refill buffers, write buffers and write-back buffers, is also counted.

CP15 cache maintenance operations do not count as events.

The PMU architecture mnemonic for this event is L1D_CACHE.

0x05, Level 1 data TLB refill

The counter counts each memory-read or memory-write operation that causes a TLB refill of at least the Level 1 data or unified TLB. Each read or write that causes a refill, in the form of a translation table walk or an access to another level of TLB caching is counted.

CP15 TLB maintenance operations do not count as events.

The PMU architecture mnemonic for this event is L1D_TLB_REFILL.

0x10, Mispredicted or not predicted branch speculatively executed

The counter counts for each correction to the predicted program flow that occurs because of a misprediction from, or no prediction from, the branch prediction resources and that relates to instructions that the branch prediction resources are capable of predicting.

The PMU architecture mnemonic for this event is BR_MIS_PRED.

0x11, Cycle The counter increments on every cycle.

Note

Unlike PMCCNTR, this count is not affected by PMCR.DP, PMCR.D or PMCR.C:

- the counter is not incremented in prohibited regions, so is not affected by PMCR.DP
- the counter increments on every cycle, regardless of the setting of PMCR.D
- the counter is reset when event counters are reset by PMCR.P, never by PMCR.C.

The PMU architecture mnemonic for this event is CPU_CYCLES.

0x12, Predictable branch speculatively executed

The counter counts every branch or other change in the program flow that the branch prediction resources are capable of predicting.

The PMU architecture mnemonic for this event is BR_PRED.

0x13, Data memory access

The counter counts memory-read or memory-write operations that the processor made.

The counter increments whether the access results in an access to a Level 1 data or unified cache, a Level 2 data or unified cache, or neither of these. The counter does not increment as a result of:

- instruction prefetches
- translation table walks
- write-back from any cache
- refilling of any cache

The PMU architecture mnemonic for this event is MEM_ACCESS.

0x14, Level 1 instruction cache access

The counter counts instruction memory accesses that access at least Level 1 instruction or unified cache. Each access to other Level 1 instruction memory structures such as refill buffers, is also counted.

The PMU architecture mnemonic for this event is L1I_ICACHE.

0x15, Level 1 data cache write-back

The counter counts every write-back of data from the Level 1 data or unified cache. The counter counts each write-back that causes data to be written from the Level 1 cache to outside of the Level 1 cache. For example, the counter counts the following cases:

- A write-back that causes data to be written to a Level 2 cache or memory.
- A write-back of a recently fetched cache line that has not been allocated to the cache.
- Transfer of data from this cache to outside of this cache made as a result of a coherency request. The conditions to which of these are counted for transfers to other Level 1 caches within the same multiprocessor cluster are IMPLEMENTATION DEFINED.

Whether this also includes write-backs made as a result of CP15 cache maintenance operations is IMPLEMENTATION DEFINED.

Note

- Invalidation of a cache line without any write-back to a Level 2 cache or memory is not counted.
- Writes from the processor that write through the Level 1 cache to outside the Level 1 cache are not counted.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

The PMU architecture mnemonic for this event is L1D_CACHE_WB.

`0x16`, **Level 2 data cache access**

The counter counts memory-read or memory-write operations that access at least the Level 2 data or unified cache. Each access to a cache line is counted including refills of and write-backs from the Level 1 caches. Each access to other Level 2 data or unified memory structures such as refill buffers, write buffers and write-back buffers, is also counted.

CP15 cache maintenance operations do not count as events. Operations made by other processors that share this cache do not count as events.

The PMU architecture mnemonic for this event is L2D_CACHE_ACCESS.

`0x17`**, Level 2 data cache refill**

The counter counts memory-read or write operations that access at least the Level 2 data or unified cache and cause a refill of a Level 1 cache or of the Level 2 data or unified cache. Each read from or write to the cache that causes a refill from outside the cache is counted. Accesses that do not cause a new cache refill, but are satisfied from refilling data of a previous miss are not counted.

Each access to a cache line that causes a new linefill is counted including refills of, and write-backs from, a Level 1 cache. Accesses to a cache line that generate a memory access but not a new linefill, such as Write-through writes that hit in the cache, are not counted.

CP15 cache maintenance operations do not count as events. Operations made by other processors that share this cache do not count as events.

The PMU architecture mnemonic for this event is L2_DCACHE_REFILL.

`0x18`**, Level 2 data cache write-back**

The counter counts every write-back of data from the Level 2 data or unified cache that the processor made. Each write-back that causes data to be written from the Level 2 cache to outside the Level 1 and Level 2 caches is counted. For example, each of the following cases is counted:

- A write back that causes data to be written to a Level 3 cache or memory.
- A write-back of a recently fetched cache line that has not been allocated to the cache.
- A transfer of data from this cache to outside the Level 1 or Level 2 cache made as a result of a coherency request. The conditions under which transfers of this type are counted for transfers to other Level 2 caches within the same multiprocessor cluster are IMPLEMENTATION DEFINED.

Whether this includes write-backs made as a result of CP15 cache maintenance operations is IMPLEMENTATION DEFINED.

Note

- Invalidation of a cache line without any write back to a Level 3 cache or memory is not counted.

- Writes from the processor or Level 1 cache that write through the Level 2 cache to outside the Level 1 and Level 2 caches are not counted.
- Transfers of data from this cache to a Level 1 cache to satisfy a Level 1 cache refill are not counted.

Each write-back is counted once, even if multiple accesses are required to complete the write-back.

The PMU architecture mnemonic for this event is L2_DCACHE_WB.

`0x19`, Bus access

The counter counts memory-read or memory-write operations that access outside of the boundary of the processor and its closely-coupled caches. Where this boundary lies with respect to any implemented caches is IMPLEMENTATION DEFINED. It must count accesses beyond the cache furthest from the processor for which accesses can be counted. The definition of a bus access is IMPLEMENTATION DEFINED, but physically is a single beat rather than a burst. That is, for each bus cycle for which the bus is active.

This means that:

- If Level 2 cache access events are implemented and no IMPLEMENTATION DEFINED events can count accesses for any caches outside a Level 2 cache, this counter increments for an access beyond the Level 2 cache.
- If Level 2 cache access events are not implemented and Level 1 cache access events are implemented, this counter increments for an access beyond the Level 1 cache.
- If neither Level 1 or Level 2 cache access events are implemented, this counter increments for all data accesses that the processor made.

Where an implementation has multiple external buses, this event counts the sum of accesses across all buses. If a bus supports multiple accesses per cycle, for example through multiple channels, the counter increments once for each channel that is active on a cycle, and so it might increment by more than one in any given cycle.

The PMU architecture mnemonic for this event is BUS_ACCESS.

`0x1A`, Local memory error

The counter counts every occurrence of a memory error signaled by a memory closely coupled to this processor. The definition of local memories is IMPLEMENTATION DEFINED, but typically includes caches, tightly-coupled memories, and TLB arrays.

Memory error refers to a physical error detected by the hardware such as a parity error. It includes both errors that are correctable and those that are not. It does not include errors as defined in the architecture such as MMU and MPU faults.

The PMU architecture mnemonic for this event is MEMORY_ERROR.

`0x1B`**, Instruction speculatively executed**

The counter counts instructions that are speculatively executed by the processor. This includes instructions that are subsequently not architecturally executed. As a result, this event counts a larger number of instructions than the number of instructions architecturally executed. The definition of speculatively executed is IMPLEMENTATION DEFINED.

The PMU architecture mnemonic for this event is INSTR_SPEC.

`0x1D`**, Bus cycle**

The register is incremented on every cycle of the external memory interface of the processor.

Note

If the processor clocks the external memory interface at the same rate as the processor, the counter counts every cycle.

The PMU architecture mnemonic for this event is BUS_CYCLES.

2.5.5 IMPLEMENTATION DEFINED feature event numbers

For IMPLEMENTATION DEFINED feature numbers, each counter is defined, independently, to either:

- increment only once for each event
- count the duration for which an event occurs.

ARM recommends that implementers establish a standardized numbering scheme for their IMPLEMENTATION DEFINED events, with common definitions, and common count numbers, applied to all the processors they implement. In general, the recommended approach is for standardization across implementations with common features. However, ARM recognizes that attempting to standardize the encoding of microarchitectural features across too wide a range of implementations is not productive.

ARM strongly recommends that at least the following classes of event are identified in the IMPLEMENTATION DEFINED events:

- Cumulative duration of stalls resulting from the holes in the instruction availability, separating out counts for key buffering points that might exist.
- Cumulative duration data-dependent stalls, separating out counts for key dependency classes that might exist.
- Cumulative duration of stalls due to unavailability of execution resources, including, for example, write buffers, separating out counts for key resources that might exist.
- Missed superscalar issue opportunities, if relevant, separating out counts for key classes of issue that might exist.
- Miss rates for different levels of caches and TLBs.
- Any external events passed to the processor through an IMPLEMENTATION DEFINED mechanism.

- Cumulative durations for which the CPSR.I and CPSR.F interrupt mask bits are set to 1.
- Any other microarchitectural features that the implementer considers it valuable to count.

IMPLEMENTATION DEFINED feature numbers are `0x40` to `0xFF`. *ARM recommendations for implementation defined event numbers* gives the ARM recommended standardized numbering scheme for these events.

ARM recommendations for IMPLEMENTATION DEFINED event numbers

ARM is developing recommendations for the assignment of the IMPLEMENTATION DEFINED event numbers. ARM does not define these events as rigorously as those in the architectural and microarchitectural event lists, and an implementation might:

- modify the definition of an event to better correspond to the implementation
- not use some, or many, of these event numbers.

Note

As recommendations, these do not form part of the PMU architecture. Contact ARM for more information about the recommendations.

2.6 Performance monitor registers

This section describes the optional performance monitor registers.

2.6.1 CP15 c9 register map

The performance monitor registers are part of the CP15 register map. Figure 2-1 shows the CP15 c9 encodings for the recommended performance monitor registers, and the reserved encodings for IMPLEMENTATION DEFINED performance monitors:

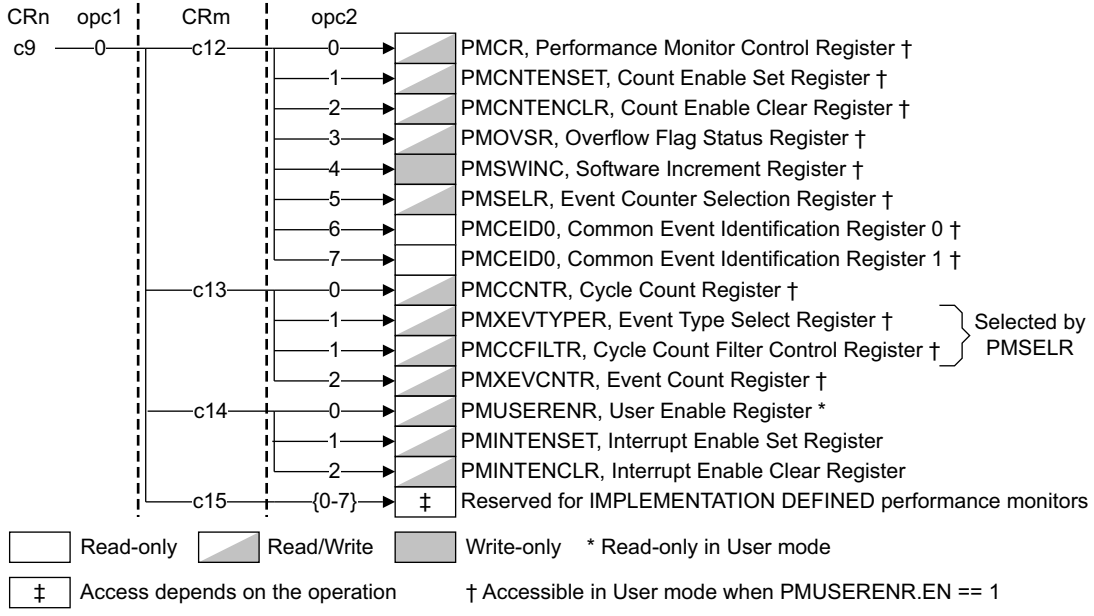


Figure 2-1 Recommended CP15 performance monitor registers

See *Access permissions, PMUv2* on page 2-39 for information about User mode access to the performance monitors.

Table 2-6 on page 2-38 shows the recommended performance monitor registers, and where each register is described in full.

Table 2-6 Performance monitor register summary

opc1	CRm[a]	opc2	Name	Type	Description
0	c12	0	PMCR	RW	*CP15 c9, Performance Monitor Control Register, PMCR* on page 2-40.
		1	PMCNTENSET	RW	*CP15 c9, Count Enable Set Register, PMCNTENSET* on page 2-44.
		2	PMCNTENCLR	RW	*CP15 c9, Count Enable Clear Register, PMCNTENCLR* on page 2-45.
		3	PMOVSR	RW	*CP15 c9, Overflow Flag Status Register, PMOVSR* on page 2-47.
		4	PMSWINC	WO	*CP15 c9, Software Increment Register, PMSWINC* on page 2-48.
		5	PMSELR	RW	*CP15 c9, Event Counter Selection Register, PMSELR* on page 2-49.
		6	PMCEID0	RO	*CP15 c9, Common Event Identification Registers 0 and 1, PMCEID0 and PMCEID1* on page 2-51.
		7	PMCEID1	RO	
	c13	0	PMCCNTR	RW	*CP15 c9, Cycle Count Register, PMCCNTR* on page 2-53.
		1	PMXEVTYPER	RW	*CP15 c9, Event Type Select Register, PMXEVTYPER* on page 2-55.
			PMCCFILTR	RW	*CP15 c9, Cycle Count Filter Control Register, PMCCFILTR* on page 2-58.
		2	PMXEVCNTR	RW	*CP15 c9, Event Count Register, PMXEVCNTR* on page 2-60.
	c14	0	PMUSERENR	RW	*CP15 c9, User Enable Register, PMUSERENR* on page 2-61.
		1	PMINTENSET	RW	*CP15 c9, Interrupt Enable Set Register, PMINTENSET* on page 2-62.
		2	PMINTENCLR	RW	*CP15 c9, Interrupt Enable Clear Register, PMINTENCLR* on page 2-64.

a. CP15 c9 encodings with CRm == {c12-c14} not listed in the table are reserved. For details of the behavior of accesses to these encodings see the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

Power domains and performance monitor registers reset

ARM recommends that performance monitors are implemented as part of the core power domain, not as part of a separate debug power domain. There is no interface to access the performance monitor registers when the core power domain is powered down.

A Reset exception sets the performance monitor registers to their reset values. A debug logic reset does not change the values of the performance monitor registers.

For more information about the reset scheme recommended for a v7 Debug implementation see the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

Access permissions, PMUv2

Normally the performance monitor registers are accessible from Secure and Non-secure privileged modes. However, a debugger can modify the access permissions for the PMU registers by setting the PMUSERENR.EN bit to 1 to permit access from User mode. Table 2-7 shows the PMU register access permissions. In the table, the following terms describe the behavior of the accesses:

Proceed The behavior on reads or writes further depends on what counter access is granted for each counter in the current mode and state.

UNDEFINED An Undefined Instruction exception is generated.

UNPREDICTABLE

The behavior is UNPREDICTABLE.

Table 2-7 Performance monitor access permissions, PMUv2

Register	Operation	Privileged access	Unprivileged access[a]	
			PMUSERENR.EN = 0	PMUSERENR.EN = 1
PMCR	MRC or MCR	Proceed	UNDEFINED	Proceed
PMCNTENSET	MRC or MCR	Proceed	UNDEFINED	Proceed
PMCNTENCLR	MRC or MCR	Proceed	UNDEFINED	Proceed
PMOVSR	MRC or MCR	Proceed	UNDEFINED	Proceed
PMSWINC	MCR	Proceed	UNDEFINED	Proceed
PMSELR	MRC or MCR	Proceed	UNDEFINED	Proceed
PMCEID0, PMCEID1	MRC	Proceed	UNDEFINED	Proceed
	MCR	UNPREDICTABLE	UNDEFINED	UNPREDICTABLE
PMCCNTR	MRC or MCR	Proceed	UNDEFINED	Proceed

Table 2-7 Performance monitor access permissions, PMUv2 (continued)

Register	Operation	Privileged access	Unprivileged access[a]	
			PMUSERENR.EN = 0	PMUSERENR.EN = 1
PMXEVTYPER, PMCCFILTR	MRC or MCR	Proceed	UNDEFINED	Proceed
PMXEVCNTR	MRC or MCR	Proceed	UNDEFINED	Proceed
PMUSERENR	MRC	Proceed	Proceed	Proceed
	MCR	Proceed	UNDEFINED	UNDEFINED
PMINTENSET	MRC or MCR	Proceed	UNDEFINED	UNDEFINED
PMINTENCLR	MRC or MCR	Proceed	UNDEFINED	UNDEFINED
Reserved[b]	MRC or MCR	UNPREDICTABLE	UNDEFINED	UNDEFINED

a. For details of the PMUSERENR.EN flag see *CP15 c9, User Enable Register, PMUSERENR* on page 2-61.
b. The registers shown as Reserved in Table 2-6 on page 2-38.

Reserved registers

Table 2-7 on page 2-39 describes the access permissions for reserved registers. This includes read accesses to write-only registers and write accesses to read-only registers, and applies to:

- all accesses to reserved register encodings
- MRC accesses to PMSWINC
- in PMUv2, MCR accesses to PMCEID0
- in PMUv2, MCR accesses to PMCEID1.

2.6.2 CP15 c9, Performance Monitor Control Register, PMCR

The PMCR characteristics are:

Purpose Provides details of the performance monitor implementation, including the number of counters implemented, and configures and controls the counters.

Usage constraints Accessible in:

- Secure and Non-secure privileged modes
- User mode only when the PMUSERENR.EN bit is set to 1, see *Access permissions, PMUv2* on page 2-39.

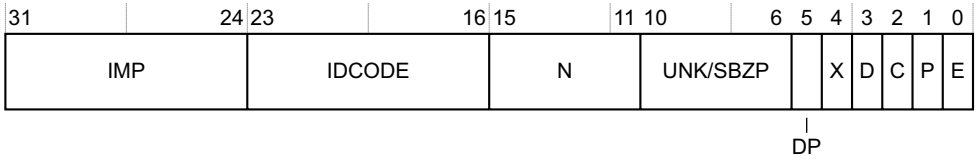
Configurations Optional. Implemented only as part of the recommended performance monitors.

In a VMSA implementation that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table 2-6 on page 2-38.

The reset value depends on the register implementation. For more information see the register bit descriptions and *Power domains and performance monitor registers reset* on page 2-39.

The PMCR bit assignments



are:

IMP, bits[31:24]

Implementer code. This is a read-only field with an IMPLEMENTATION DEFINED value.

The Implementer codes are allocated by ARM. Values have the same interpretation as bits[31:24] of the CP15 Main ID Register, see the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition.*

IDCODE, bits[23:16]

Identification code. This is a read-only field with an IMPLEMENTATION DEFINED value.

Each implementer must maintain a list of identification codes that is specific to the implementer. A specific implementation is identified by the combination of the implementer code and the identification code.

N, bits[15:11]

Number of event counters. This is a read-only field with an IMPLEMENTATION DEFINED value that indicates the number of counters implemented.

The value of this field is the number of counters implemented, from 0b00000 for no counters to 0b11111 for 31 counters.

An implementation can implement only the Cycle Count Register, PMCCNTR. This is indicated by a value of 0b00000 for the N field.

Bits[10:6] Reserved, UNK/SBZP.

DP, bit[5] Disable PMCCNTR when prohibited. The possible values of this bit are:

0 Count is enabled in prohibited regions

1 Count is disabled in prohibited regions.

Prohibited regions are defined as regions where event counting would be prohibited. For example, if non-invasive debug is disabled in all Secure modes, the Secure state is a prohibited region. For details of non-invasive debug authentication see the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

Note

A Non-secure process can set this bit to 1, to discard cycle counts that might be accumulated during periods when the other counts are prohibited because of security prohibitions. It is not a control to enhance security. The function of this bit is to avoid corruption of the count. See also *Effect of the Security Extensions* on page 2-19.

This is a read/write bit. Its core logic reset value is 0.

X, bit[4] Export enable. The possible values of this bit are:

0 Export of events is disabled

1 Export of events is enabled.

This bit is used to permit events to be exported to another debug device, such as a trace macrocell, over an event bus. If the implementation does not include such an event bus, this bit is RAZ/WI.

This bit does not affect the generation of performance monitor interrupts, that can be implemented as a signal exported from the processor to an interrupt controller.

This is a read/write bit. Its core logic reset value is 0.

D, bit[3] Clock divider. The possible values of this bit are:

0 When enabled, PMCCNTR counts every clock cycle

1 When enabled, PMCCNTR counts once every 64 clock cycles.

This is a read/write bit. Its core logic reset value is 0.

C, bit[2] Clock counter reset. This is a write-only bit. The effects of writing to this bit are:

0 No action

1 Reset PMCCNTR to zero.

Note

Resetting PMCCNTR does not clear the PMCCNTR overflow bit to 0. For more information see *CP15 c9, Overflow Flag Status Register, PMOVSR* on page 2-47.

This bit is always RAZ.

P, bit[1] Event counter reset. This is a write-only bit. The effects of writing to this bit are:

0 No action

1 Reset all event counters, not including PMCCNTR, to zero.

Note

Resetting the event counters does not clear any overflow bits to 0. For more information see *CP15 c9, Overflow Flag Status Register, PMOVSR* on page 2-47.

This bit is always RAZ.

E, bit[0] Enable. The possible values of this bit are:

0 All counters, including PMCCNTR, are disabled

1 All counters are enabled.

Performance monitor overflow IRQs are only signaled when the enable bit is set to 1.

This is a read/write bit. Its core logic reset value is 0.

Accessing the PMCR

To access PMCR, read or write the CP15 registers with <opc1> set to 0, <CRn> set to c9, <CRm> set to c12, and <opc2> set to 0. For example:

```
MRC p15,0,<Rt>,c9,c12,0    ; Read Performance Monitor Control Register
MCR p15,0,<Rt>,c9,c12,0    ; Write Performance Monitor Control Register
```

2.6.3 CP15 c9, Count Enable Set Register, PMCNTENSET

The PMCNTENSET Register characteristics are:

Purpose Enables the Cycle Count Register, PMCCNTR, and any implemented event counters, PMN*x*. Reading this register shows which counters are enabled.

Usage constraints Used in conjunction with the PMCNTENCLR, see *CP15 c9, Count Enable Clear Register, PMCNTENCLR* on page 2-45

Accessible in:

- Secure and Non-secure privileged modes
- User mode only when the PMUSERENR.EN bit is set to 1, see *Access permissions, PMUv2* on page 2-39.

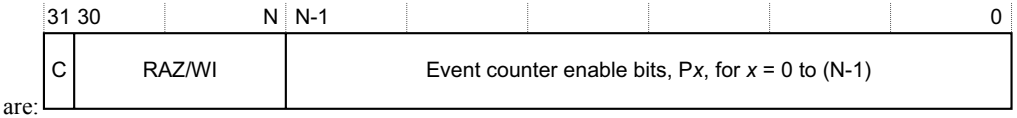
Configurations Optional. Implemented only as part of the recommended performance monitors.

In a VMSA implementation that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table 2-6 on page 2-38.

The contents of the PMCNTENSET Register are UNKNOWN on a core logic reset. See also *Power domains and performance monitor registers reset* on page 2-39.

The PMCNTENSET Register bit assignments are:



Note

In the description of the PMCNTENSET Register:

- N is the number of event counters implemented, as defined by the PMCR.N field, see *CP15 c9, Performance Monitor Control Register, PMCR* on page 2-40
- *x* refers to a single event counter, and takes values from 0 to (N-1).

C, bit[31] PMCCNTR enable bit.

See Table 2-8 on page 2-45 for the behavior of this bit on reads and writes.

Bits[30:N] RAZ/WI.

P*x*, bit[*x*], for *x* = 0 to (N-1)

Event counter *x*, PMN*x*, enable bit.

Table 2-8 shows the behavior of this bit on reads and writes.

Table 2-8 Read and write bit values for the PMCNTENSET Register

Value	Meaning on read	Action on write
0	Counter disabled	No action, write is ignored
1	Counter enabled	Enable counter

Accessing the PMCNTENSET Register

To access the PMCNTENSET Register, read or write the CP15 registers with <opc1> set to 0, <CRn> set to c9, <CRm> set to c12, and <opc2> set to 1. For example:

```
MRC p15,0,<Rt>,c9,c12,1    ; Read Count Enable Set Register
MCR p15,0,<Rt>,c9,c12,1    ; Write Count Enable Set Register
```

2.6.4 CP15 c9, Count Enable Clear Register, PMCNTENCLR

The PMCNTENCLR Register characteristics are:

Purpose Disables the Cycle Count Register, PMCCNTR, and any implemented event counters, PMN*x*. Reading this register shows which counters are enabled.

Usage constraints Used in conjunction with the PMCNTENSET Register, see *CP15 c9, Count Enable Set Register, PMCNTENSET* on page 2-44

Accessible in:

- Secure or Non-secure privileged modes
- User mode only when the PMUSERENR.EN bit is set to 1, see *Access permissions, PMUv2* on page 2-39.

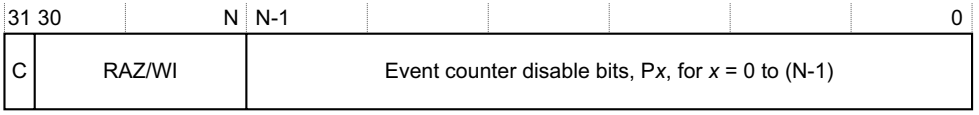
Configurations Optional. Implemented only as part of the recommended performance monitors.

In a VMSA implementation that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table 2-6 on page 2-38.

The contents of the PMCNTENCLR Register are UNKNOWN on a core logic reset. See also *Power domains and performance monitor registers reset* on page 2-39.

The PMCNTENCLR Register bit assignments are:



Note

In the description of the PMCNTENCLR Register, N and *x* have the meanings used in the description of the PMCNTENSET Register, see *CP15 c9, Count Enable Set Register, PMCNTENSET* on page 2-44.

C, bit[31] PMCCNTR disable bit.

See Table 2-9 for the behavior of this bit on reads and writes.

Bits[30:N] RAZ/WI.

P*x*, bit[*x*], for *x* = 0 to (N-1)

Event counter *x*, PMN*x*, disable bit.

Table 2-9 shows the behavior of this bit on reads and writes.

Table 2-9 Read and write bit values for the PMCNTENCLR Register

Value	Meaning on read	Action on write
0	Counter disabled	No action, write is ignored
1	Counter enabled	Disable counter

Note

The PMCR.E Enable bit can be used to override the settings in this register and disable all counters including PMCCNTR, see *CP15 c9, Performance Monitor Control Register, PMCR* on page 2-40. The counter enable register retains its value when the Enable bit is 0, even though its settings are ignored.

Accessing the PMCNTENCLR Register

To access the PMCNTENCLR Register, read or write the CP15 registers with <opc1> set to 0, <CRn> set to c9, <CRm> set to c12, and <opc2> set to 2. For example:

```
MRC p15,0,<Rt>,c9,c12,2    : Read Count Enable Clear Register
MCR p15,0,<Rt>,c9,c12,2    : Write Count Enable Clear Register
```

2.6.5 CP15 c9, Overflow Flag Status Register, PMOVSR

The PMOVSR characteristics are:

Purpose Holds the state of the overflow bits for the Cycle Count Register, PMCCNTR, and each of the implemented event counters, PMN*x*. Software must write to this register to clear these bits.

Usage constraints Accessible in:

- Secure and Non-secure privileged modes
- User mode only when the PMUSERENR.EN bit is set to 1, see *Access permissions, PMUv2* on page 2-39.

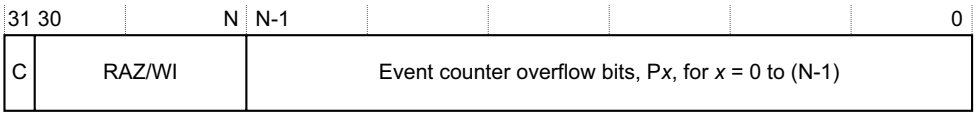
Configurations Optional. Implemented only as part of the recommended performance monitors.

In a VMSA implementation that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table 2-6 on page 2-38.

The contents of the PMCNTENCLR Register are UNKNOWN on a core logic reset. See also *Power domains and performance monitor registers reset* on page 2-39.

The PMOVSR bit assignments are:



Note

In the description of the PMOVSR, N and *x* have the meanings used in the description of the PMCNTENSET Register, see *CP15 c9, Count Enable Set Register, PMCNTENSET* on page 2-44.

C, bit[31] PMCCNTR overflow bit.

Table 2-10 on page 2-48 shows the behavior of this bit on reads and writes.

Bits[30:N] RAZ/WI.

P*x*, bit[*x*], for *x* = 0 to (N-1)

Event counter *x*, PMN*x*, overflow bit.

Table 2-10 shows the behavior of this bit on reads and writes.

Table 2-10 Read and write bit values for the PMOVSR

Value	Meaning on read	Action on write
0	Counter has not overflowed	No action, write is ignored
1	Counter has overflowed	Clear bit to 0

The contents of the PMOVSR are UNKNOWN on a core logic reset.

Note

The overflow bit values for individual counters are retained until cleared to 0 by a write to the PMOVSR or processor reset, even if the counter is later disabled by writing to the PMCNTENCLR register or through the PMCR.E Enable bit. The overflow bits are also not cleared to 0 when the counters are reset through the Event counter reset or Clock counter reset bits in the PMCR.

Accessing the PMOVSR

To access the PMOVSR, read or write the CP15 registers with <opc1> set to 0, <CRn> set to c9, <CRm> set to c12, and <opc2> set to 3. For example:

```
MRC p15,0,<Rt>,c9,c12,3;    Read Overflow Flag Status Register
MCR p15,0,<Rt>,c9,c12,3;    Write Overflow Flag Status Register
```

2.6.6 CP15 c9, Software Increment Register, PMSWINC

The PMSWINC Register characteristics are:

Purpose Increments a counter that is configured to count the Software increment event, event 0x00.

Usage constraints Accessible in:

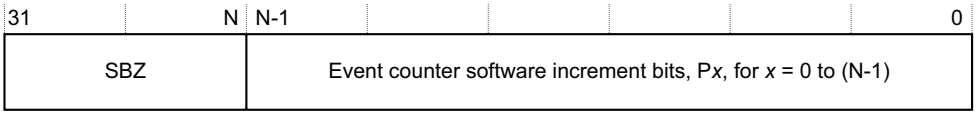
- Secure or Non-secure privileged modes
- User mode only when the PMUSERENR.EN bit is set to 1, see *Access permissions, PMUv2* on page 2-39.

Configurations Optional. Implemented only as part of the recommended performance monitors.

In a VMSA implementation that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table 2-6 on page 2-38 and *Power domains and performance monitor registers reset* on page 2-39.

The PMSWINC Register bit assignments are:



Note

In the description of the PMSWINC Register, N and *x* have the meanings used in the description of the PMCNTENSET Register, see *CP15 c9, Count Enable Set Register, PMCNTENSET* on page 2-44.

Bits[31:N] Reserved, SBZ.

P*x*, bit[*x*], for *x* = 0 to (N-1)

Event counter *x*, PMN*x*, software increment bit. This is a write-only bit. The effects of writing to this bit are:

0 No action, the write is ignored.

1, if PMN*x* is configured to count the Software increment event

Increment PMN*x* Register by 1.

1, if PMN*x* is not configured to count the Software increment event

UNPREDICTABLE.

Accessing the PMSWINC Register

To access the PMSWINC Register, write the CP15 registers with <opc1> set to 0, <CRn> set to c9, <CRm> set to c12, and <opc2> set to 4. For example:

```
MCR p15,0,<Rt>,c9,c12,4    ; Write Software Increment Register
```

2.6.7 CP15 c9, Event Counter Selection Register, PMSELR

The PMSELR characteristics are:

Purpose Selects the current event counter, PMN*x*.

Usage constraints Used in conjunction with:

- The PMXEVTYPER to determine the event that increments the selected counter. See *CP15 c9, Event Type Select Register, PMXEVTYPER* on page 2-55.
- The PMXEVCNTR to determine the value of the selected counter. See *CP15 c9, Event Count Register, PMXEVCNTR* on page 2-60.
- When SEL is set to 31, the PMCCFILTR, see *CP15 c9, Cycle Count Filter Control Register, PMCCFILTR* on page 2-58.

Accessible in Secure and Non-secure privileged modes.

Configurations Optional. Implemented only as part of the recommended performance monitors.

In a VMSA implementation that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table 2-6 on page 2-38.

The contents of the PMSELR are UNKNOWN on a core logic reset. See also *Power domains and performance monitor registers reset* on page 2-39.

The PMSELR bit assignments are:

Bits[31:5] Reserved, UNK/SBZP.

SEL, bits[4:0]

Selection value of the current event counter, PMN*x*, where *x* is the value held in this field. In addition, in PMUv2, setting this field to 31 selects the PMCCFILTR. In PMUv1, the value of 31 is Reserved and must not be used.

If this field is set to a value greater than or equal to the number of implemented counters, but less than 31, the results are UNPREDICTABLE. The number of implemented counters is defined by the PMCR.N field, see *CP15 c9, Performance Monitor Control Register, PMCR* on page 2-40.

In version 2 of the Performance monitor architecture, when PMSELR.SEL is 31:

- a read of the PMXEVTYPER returns the value of PMCCFILTR
- a write of the PMXEVTYPER writes to PMCCFILTR.
- reads and writes of PMXEVCNTR are UNPREDICTABLE.

The SEL field identifies which event counter, PMN_{SEL}, is accessed by PMXEVTYPER and PMXEVCNTR, see *CP15 c9, Event Type Select Register, PMXEVTYPER* on page 2-55 and *CP15 c9, Event Count Register, PMXEVCNTR* on page 2-60.

Accessing the PMSELR

To access the PMSELR, read or write the CP15 registers with <opc1> set to 0, <CRn> set to c9, <CRm> set to c12, and <opc2> set to 5. For example:

```
MRC p15,0,<Rt>,c9,c12,5    ; Read Event Counter Selection Register
MCR p15,0,<Rt>,c9,c12,5    ; Write Event Counter Selection Register
```

2.6.8 CP15 c9, Common Event Identification Registers 0 and 1, PMCEID0 and PMCEID1

The PMCEID0 and PMCEID1 characteristics are:

Purpose Defines which common architectural and common microarchitectural feature events are implemented.

Usage constraints The Common Event Identification Registers 0 and 1 are accessible in:

- Secure or Non-secure privileged modes
- User mode only when the PMUSERENR.EN bit is set to 1, see *Access permissions, PMUv2* on page 2-39.

Configurations Optional. Implemented only in PMUv2 as part of the recommended performance monitors.

In a VMSA implementation that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table 2-6 on page 2-38.

Table 2-11 shows the PMCEID0 bit assignments with event implemented or not implemented when the associated bit is set to 1 or 0.

PMCEID1 is reserved, UNK.

Table 2-11 Common Event Identification Register 0 bit assignments

Bit	Event number	Event[a]
[31]	0x1F	Reserved, UNK.
[30]	0x1E	
[29]	0x1D	
[28]	0x1C	Write to translation table base instruction architecturally executed, condition check pass. This bit is RAO[b].
[27]	0x1B	Instruction speculatively executed.
[26]	0x1A	Load memory error.
[25]	0x19	Bus access.
[24]	0x18	Level 2 data cache write-back.
[23]	0x17	Level 2 data cache refill.
[22]	0x16	Level 2 data cache access.
[21]	0x15	Level 1 data cache write-back.

Table 2-11 Common Event Identification Register 0 bit assignments (continued)

Bit	Event number	Event[a]
[20]	0x14	Level 1 instruction cache access.
[19]	0x13	Data memory access.
[18]	0x12	Predictable branch speculatively executed. If the processor implements branch prediction, this bit is RAO.
[17]	0x11	Cycle. This bit is RAO[b].
[16]	0x10	Mispredicted or not predicted branch speculatively executed. If the processor implements branch prediction resources, this bit is RAO.
[15]	0x0F	Unaligned load or store instruction architecturally executed, condition check pass. This bit is RAO[b].
[14]	0x0E	Procedure return instruction architecturally executed, condition check pass. This bit is RAO[b].
[13]	0x0D	Immediate branch instruction architecturally executed. This bit is RAO[b].
[12]	0x0C	Software change of the PC instruction architecturally executed, condition check pass. This bit is RAO[b].
[11]	0x0B	Write to CONTEXTIDR instruction architecturally executed, condition check pass. This bit is RAO[b].
[10]	0x0A	Exception return instruction architecturally executed, condition check pass. This bit is RAO[b].
[9]	0x09	Exception taken.
[8]	0x08	Instruction architecturally executed. This bit is RAO[b].
[7]	0x07	Store instruction architecturally executed, condition check pass. This bit is RAO[b].
[6]	0x06	Load instruction architecturally executed, condition check pass. This bit is RAO[b].
[5]	0x05	Level 1 data TLB refill.
[4]	0x04	Level 1 data cache access. If the processor implements L1 data or unified cache, this bit reads as one.
[3]	0x03	Level 1 data cache refill. If the processor implements L1 data or unified cache, this bit reads as one.

Table 2-11 Common Event Identification Register 0 bit assignments (continued)

Bit	Event number	Event[a]
[2]	0x02	Level 1 instruction TLB refill.
[1]	0x01	Level 1 instruction cache refill.
[0]	0x00	Instruction architecturally executed, condition check pass, software increment. This bit reads as one.

a. The event is supported if the bit is set to 1, and not supported if the bit is set to 0.
b. PMUv2 requires these events to be supported, and therefore the corresponding bits are RAO. PMUv1 does not have this requirement.

Accessing the PMCEID0

To access the PMCEID0, read the CP15 register with <opc1> set to 0, <CRn> set to c9, <CRm> set to c12, and <opc2> set to 6. For example:

```
MRC p15,0,<Rt>,c9,c12,6    ; Read Common Event Identification Register 0
```

Accessing the PMCEID1

To access the PMCEID1, read the CP15 register with <opc1> set to 0, <CRn> set to c9, <CRm> set to c12, and <opc2> set to 7. For example:

```
MRC p15,0,<Rt>,c9,c12,7    ; Read Common Event Identification Register 1
```

2.6.9 CP15 c9, Cycle Count Register, PMCCNTR

The PMCCNTR characteristics are:

Purpose Counts processor clock cycles.

Usage constraints The PMCR.D bit configures whether PMCCNTR increments every clock cycle, or every 64 clock cycles. See *CP15 c9, Performance Monitor Control Register, PMCR* on page 2-40.

Accessible in:

- Secure or Non-secure privileged modes
- User mode only when the PMUSERENR.EN bit is set to 1, see *Access permissions, PMUv2* on page 2-39.

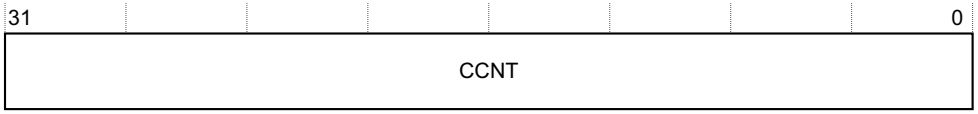
Configurations Optional. Implemented only as part of the recommended performance monitors.

In a VMSA implementation that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table 2-6 on page 2-38.

The contents of the PMCCNTR are UNKNOWN on a core logic reset. See also *Power domains and performance monitor registers reset* on page 2-39.

The PMCCNTR bit assignments are:



CCNT, bits[31:0]

Cycle count. Depending on the value of the PMCR.D bit, this field increments either:

- every processor clock cycle
- every 64th processor clock cycle.

The PMCCNTR.CCNT value can be reset to zero by writing a 1 to the PMCR.C bit, see *CP15 c9, Performance Monitor Control Register, PMCR* on page 2-40.

Accessing the PMCCNTR

To access the PMCCNTR, read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c9, `<CRm>` set to c13, and `<opc2>` set to 0. For example:

```
MRC p15,0,<Rt>,c9,c13,0    : Read Cycle Count Register
MCR p15,0,<Rt>,c9,c13,0    : Write Cycle Count Register
```

2.6.10 CP15 c9, Event Type Select Register, PMXEVTYPER

The PMXEVTYPER characteristics are:

Purpose Configures which event increments the current event counter, PMN*x*.

Usage constraints Used in conjunction with the PMSELR, see *CP15 c9, Event Counter Selection Register, PMSELR* on page 2-49.

Accessible in:

- Secure or Non-secure privileged modes
- User mode only when the PMUSERENR.EN bit is set to 1, see *Access permissions, PMUv2* on page 2-39.

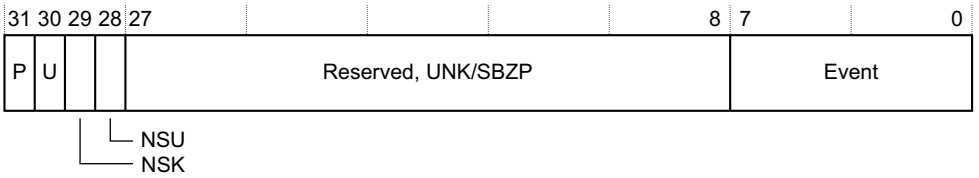
Configurations Optional. Implemented only as part of the recommended performance monitors.

In a VMSA implementation that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table 2-6 on page 2-38.

The value of each Event Type Select Register is UNKNOWN on a core logic reset. See also *Power domains and performance monitor registers reset* on page 2-39.

In PMUv2, the PMXEVTYPER bit assignments are:



P, bit[31] Privileged modes filtering bit. Controls counting in Secure privileged modes. The possible values of this bit are:

0 Count events in Secure privileged modes

1 Do not count events in Secure privileged modes.

In an implementation that does not include the Security Extension, this bit controls the counting of events in privileged modes.

The reset value of this bit is UNKNOWN.

U, bit[30] User mode filtering bit. Controls counting in Secure User mode. The possible values of this bit are:

0 Count events in Secure User mode

1 Do not count events in Secure User mode.

In an implementation that does not include the Security Extension, this bit controls the counting of events in User mode.

The reset value of this bit is UNKNOWN.

NSK, bit[29], Security Extensions implemented

Non-secure privileged mode control bit. Controls counting in Non-secure privileged modes. Most applications can ignore this bit and set the value to zero. The behavior depends on the combined values of P and NSK:

P = NSK Count events in Non-secure privileged modes

P != NSK Do not count events in Non-secure privileged modes.

The reset value of this bit is UNKNOWN.

NSK, bit[29], Security Extensions not implemented

Reserved, RAZ/WI.

NSU, bit[28], Security Extensions implemented

Non-secure User mode control bit. Control counting in Non-secure User modes. Most applications can ignore this bit and set the value to zero. The behavior depends on the combined values of U and NSU:

U = NSU Count events in Non-secure User mode

U != NSU Do not count events in Non-secure User mode.

The reset value of this bit is UNKNOWN.

NSU, bit[28], Security Extensions not implemented

Reserved, RAZ/WI.

Bits[27:8] Reserved, UNK/SBZP.

Event, bits[7:0]

Event to count. The Event number of the event that is counted by the current event counter, PMN*x*. For more information, see *Event numbers* on page 2-58.

Interpretation of the P, U, NSK, and NSU bits

In implementation that include the Security Extensions, v2 of the Performance monitors adds separate control of event counting for Secure and Non-secure modes. This additional control, provided by the NSK and NSU bits, provides backwards compatibility by ensuring that, when the NSK and NSU bits are both set to 0, the effects of the different values of the P and U bits are identical to their effects in Performance monitors v1. This mean the settings might seem non-intuitive. If, for the P, U, NSK, and NSU bits, 1 corresponds to TRUE and 0 to FALSE, Table 2-12 on page 2-57 shows the logical conditions for counting events in each of the supported combinations of mode and security state:

Table 2-12 Logic of the P, U, NSK, and NSU bit values

Count in	Condition
Secure privileged modes	If NOT P
Secure User mode	If NOT U
Non-secure privileged modes	If NOT (P AND NSK)
Non-secure User mode	If NOT (U AND NSU)

Note

Regardless of the values of the P, U, NSK and NSU bits, the performance monitors do not count any events when non-invasive debug is disabled or not permitted. For details of non-invasive debug authentication, see the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

Reserved encodings in PMXEVTYPER

Table 2-13 shows the combination of reserved encodings that software must not select. However, these are not UNPREDICTABLE encodings and hardware must implement the P, U, NSK, and NSU filtering bits as described.

Table 2-13 Reserved encodings, do not use

P	U	NSK	NSU	Modes in which events are counted
0	1	1	1	Secure privileged modes and Non-secure User mode
1	0	1	1	Secure User mode and Non-secure privileged modes
1	1	0	0	Never

Accessing the PMXEVTYPER

To access the PMXEVTYPER, read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c9, `<CRm>` set to c13, and `<opc2>` set to 1. For example:

```
MRC p15,0,<Rt>,c9,c13,1    : Read Event Type Select Register
MCR p15,0,<Rt>,c9,c13,1    : Write Event Type Select Register
```

In PMUv2, when PMSELR.SEL is 0b11111:

- a read of the PMXEVTYPER returns the value of PMCCFILTR
- a write of the PMXEVTYPER writes to PMCCFILTR.

Event numbers

The PMXEVTYPER uses event numbers to determine the event that causes an event counter to increment. These event numbers are split into two ranges:

`0x00-0x3F` Common features. Reserved for the specified events. When an ARMv7 processor supports monitoring of an event that is assigned a number in this range, if possible it must use that number for the event. Unassigned values are reserved and might be used to define additional common events in future versions of the architecture.

`0x40-0xFF` IMPLEMENTATION DEFINED features.

For more information, including the assigned values in the common features range, see *Event numbers and mnemonics* on page 2-23.

2.6.11 CP15 c9, Cycle Count Filter Control Register, PMCCFILTR

PMUv2 defines the Cycle Count Filter Control Register. The PMCCFILTR characteristics are:

Purpose Configures which modes and states the cycle counter, CCNT, counts in.

Usage constraints Used in conjunction with the PMSELR, see *CP15 c9, Event Counter Selection Register, PMSELR* on page 2-49. PMCCFILTR is accessible only when PMSELR.SEL is set to 31.

Accessible in:

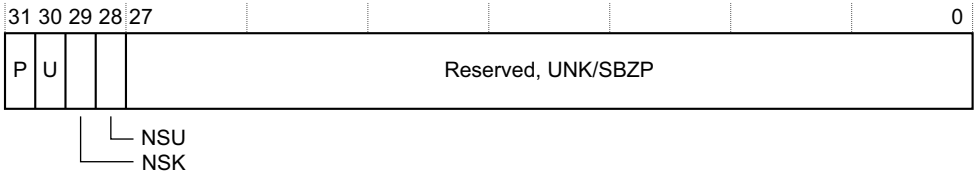
- Secure and Non-secure privileged modes
- User mode only when the PMUSERENR.EN bit is set to 1, see *Access permissions, PMUv2* on page 2-39.

Configurations Optional. Implemented only as part of the recommended performance monitors.

In a VMSA implementation that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table 2-6 on page 2-38.

The PMCCFILTR bit assignments are:



P, bit[31] Privileged modes filtering bit. Controls counting in Secure privileged modes. The possible values of this bit are:

0 Count events in Secure privileged modes

1 Do not count events in Secure privileged modes.

The reset value of this bit is 0.

U, bit[30] User mode filtering bit. Controls counting in Secure User mode. The possible values of this bit are:

0 Count events in Secure User mode

1 Do not count events in Secure User mode.

The reset value of this bit is 0.

NSK, bit[29], Security Extensions implemented

Non-secure privileged mode control bit. Controls counting in Non-secure privileged modes. Most applications can ignore this bit and set the value to zero. The behavior depends on the combined values of P and NSK:

P = NSK Count events in Non-secure privileged modes

P != NSK Do not count events in Non-secure privileged modes.

The reset value of this bit is 0.

NSK, bit[29], Security Extensions not implemented

Reserved, RAZ/WI.

NSU, bit[28], Security Extensions implemented

Non-secure User mode control bit. Control counting in Non-secure User modes. Most applications can ignore this bit and set the value to zero. The behavior depends on the combined values of U and NSU:

U = NSU Count events in Non-secure User mode

U != NSU Do not count events in Non-secure User mode.

The reset value of this bit is 0.

NSU, bit[28], Security Extensions not implemented

Reserved, RAZ/WI.

Bits[27:0] Reserved, UNK/SBZP.

Note

The Cycle Count Filter Control Register, PMCCFILTR, is not defined in the PMU architecture version 1 so existing software does not write to it before enabling the cycle counter. PMCCFILTR differs from PMXEVTYPER in that it must be initialized to enable an event, therefore the equivalent bits in PMXEVTYPER have no defined reset value.

Accessing the PMCCFILTR

To access the PMCCFILTR, read or write the CP15 registers with <opc1> set to 0, <CRn> set to c9, <CRm> set to c13, <opc2> set to 1, and PMSELR.SEL set to 0b11111. For example:

```
MRC p15,0,<Rt>,c9,c13,1    : Read Cycle Count Filter Control Register
MCR p15,0,<Rt>,c9,c13,1    : Write Cycle Count Filter Control Register
```

In PMUv2, when PMSELR.SEL is 0b11111:

- a read of the PMXEVTYPER returns the value of PMCCFILTR
- a write of the PMXEVTYPER writes to PMCCFILTR.

2.6.12 CP15 c9, Event Count Register, PMXEVCNTR

The PMXEVCNTR characteristics are:

Purpose Used to read or write the value of the current event counter, PMN*x*.

Usage constraints Used in conjunction with the PMSELR, see *CP15 c9, Event Counter Selection Register, PMSELR* on page 2-49.

Accessible in:

- Secure and Non-secure privileged modes
- User mode only when the PMUSERENR.EN bit is set to 1, *Access permissions, PMUv2* on page 2-39.

When PMSELR selects a counter for which an access is not permitted in the current mode and state:

- a read of PMXEVCNTR returns an UNKNOWN value
- the effect of a write to PMXEVCNTR is UNPREDICTABLE.

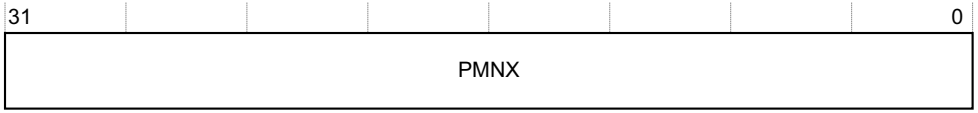
Configurations Optional. Implemented only as part of the recommended performance monitors.

In a VMSA implementation that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table 2-6 on page 2-38.

The value of each Event Count Register is UNKNOWN on a core logic reset. See also *Power domains and performance monitor registers reset* on page 2-39.

The PMXEVCNTR bit assignments are:



PMNX, bits[31:0]

Value of the current event counter, PMN*x*.

Note

Software can write to the PMXEVCNTR even when the counter is disabled. This is true regardless of why the counter is disabled, which can be any of:

- because 1 has been written to the appropriate bit in the PMCNTENCLR
- because the PMCR.E bit is set to 0
- by the non-invasive debug authentication.

Accessing the PMXEVCNTR

To access the PMXEVCNTR, read or write the CP15 registers with `<opc1>` set to 0, `<CRn>` set to c9, `<CRm>` set to c13, and `<opc2>` set to 2. For example:

```
MRC p15,0,<Rt>,c9,c13,2    : Read Event Count Register
MCR p15,0,<Rt>,c9,c13,2    : Write Event Count Register
```

2.6.13 CP15 c9, User Enable Register, PMUSERENR

The PMUSERENR characteristics are:

Purpose Enables or disables User mode access to the performance monitors.

Usage constraints Accessible in:

- Secure or Non-secure privileged modes
- read-only in User mode, see *Access permissions, PMUv2* on page 2-39.

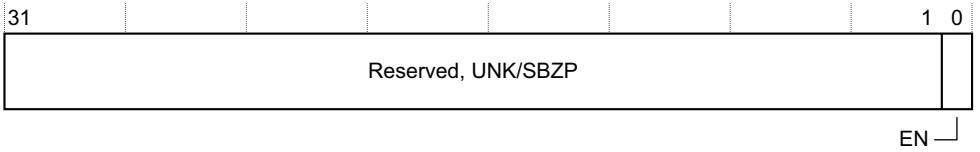
Configurations Optional. Implemented only as part of the recommended performance monitors.

In a VMSA implementation that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table 2-6 on page 2-38.

The PMUSERENR.EN bit is set to 0 on a core logic reset. See also *Power domains and performance monitor registers reset* on page 2-39.

The PMUSERENR bit assignments are:



Bits[31:1] Reserved, UNK/SBZP.

EN, bit[0] User mode access enable bit. The possible values of this bit are:

0 User mode access to performance monitors disabled

1 User mode access to performance monitors enabled.

Some MCR and MRC instructions used to access the performance monitors are UNDEFINED in User mode when User mode access to the performance monitors is disabled. For more information, see *Access permissions, PMUv2* on page 2-39.

Accessing the PMUSERENR

To access the PMUSERENR, read or write the CP15 registers with <opc1> set to 0, <CRn> set to c9, <CRm> set to c14, and <opc2> set to 0. For example:

```
MRC p15,0,<Rt>,c9,c14,0    : Read User Enable Register
MCR p15,0,<Rt>,c9,c14,0    : Write User Enable Register
```

2.6.14 CP15 c9, Interrupt Enable Set Register, PMINTENSET

The PMINTENSET Register characteristics are:

Purpose Enables the generation of interrupt requests on overflows from:

- the Cycle Count Register, PMCCNTR
- each implemented event counter, PMN*x*.

Reading the register shows which overflow interrupt requests are enabled.

Usage constraints Used in conjunction with the PMINTENCLR Register, see *CP15 c9, Interrupt Enable Clear Register, PMINTENCLR* on page 2-64.

Accessible in Secure or Non-secure privileged modes.

Instructions that access the register are always UNDEFINED in User mode, even if the PMUSERENR.EN bit is set to 1, see *CP15 c9, User Enable Register, PMUSERENR* on page 2-61.

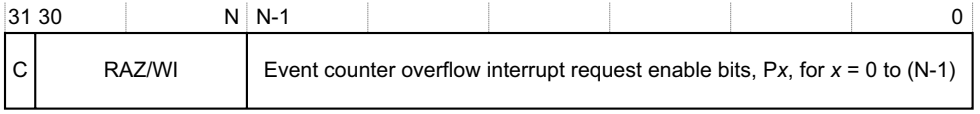
Configurations Optional. Implemented only as part of the recommended performance monitors.

In a VMSA implementation that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table 2-6 on page 2-38.

The contents of the PMINTENSET Register are UNKNOWN on a core logic reset. See also *Power domains and performance monitor registers reset* on page 2-39.

The PMINTENSET Register bit assignments are:



Note

In the description of the PMINTENSET Register, N and *x* have the meanings used in the description of the PMCNTENSET Register, see *CP15 c9, Count Enable Set Register, PMCNTENSET* on page 2-44.

C, bit[31] PMCCNTR overflow interrupt request enable bit.

See Table 2-14 for the behavior of this bit on reads and writes.

Bits[30:N] RAZ/WI.

P*x*, bit[*x*], for *x* = 0 to (N-1)

Event counter *x*, PMN*x*, overflow interrupt request enable bit.

Table 2-14 shows the behavior of this bit on reads and writes.

Table 2-14 Read and write bit values for the PMINTENSET Register

Value	Meaning on read	Action on write
0	Interrupt request disabled	No action, write is ignored
1	Interrupt request enabled	Enable interrupt request

To avoid spurious interrupts being generated, software must set the overflow interrupt request enable values before enabling any of the counters. The debug logic does not signal an interrupt request if the PMCR.E enable bit is set to 0.

When an interrupt is signaled, software can remove it by clearing the overflow bit for the counter in the PMOVSR Register to 0, see *CP15 c9, Overflow Flag Status Register, PMOVSR* on page 2-47.

Note

ARM expects that the interrupt request that can be generated on a counter overflow is also exported from the processor, meaning it can be factored into a system interrupt controller if applicable. This means that normally the system will have more levels of control of the interrupt generated.

Accessing the PMINTENSET Register

To access the PMINTENSET Register, read or write the CP15 registers with <opc1> set to 0, <CRn> set to c9, <CRm> set to c14, and <opc2> set to 1. For example:

```
MRC p15,0,<Rt>,c9,c14,1    : Read Interrupt Enable Set Register
MCR p15,0,<Rt>,c9,c14,1    : Write Interrupt Enable Set Register
```

2.6.15 CP15 c9, Interrupt Enable Clear Register, PMINTENCLR

The PMINTENCLR Register characteristics are:

Purpose Disables the generation of interrupt requests on overflows from:

- the Cycle Count Register, PMCCNTR
- each implemented event counter, PMN*x*.

Reading the register shows which overflow interrupt requests are enabled.

Usage constraints Used in conjunction with the PMINTENSET Register, see *CP15 c9, Interrupt Enable Set Register, PMINTENSET* on page 2-62.

Accessible in Secure or Non-secure privileged modes.

Instructions that access the register are always UNDEFINED in User mode, even if the PMUSERENR.EN bit is set to 1, see *CP15 c9, User Enable Register, PMUSERENR* on page 2-61.

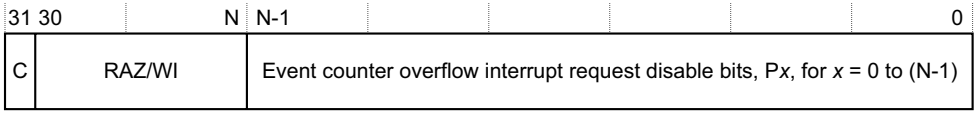
Configurations Optional. Implemented only as part of the recommended performance monitors.

In a VMSA implementation that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table 2-6 on page 2-38.

The contents of the PMINTENCLR Register are UNKNOWN on a core logic reset. See also *Power domains and performance monitor registers reset* on page 2-39.

The PMINTENCLR Register bit assignments are:



Note

In the description of the PMINTENCLR Register, N and *x* have the meanings used in the description of the PMINTENSET Register, see *CP15 c9, Interrupt Enable Set Register, PMINTENSET* on page 2-62.

C, bit[31] PMCCNTR overflow interrupt request disable bit.

See Table 2-15 on page 2-65 for the behavior of this bit on reads and writes.

Bits[30:N] RAZ/WI.

P*m*, bit[*x*], for *x* = 0 to (N-1)

Event counter *x*, PMN*x*, overflow interrupt request disable bit.

Table 2-15 shows the behavior of this bit on reads and writes.

Table 2-15 Read and write bit values for the PMINTENCLR Register

Value	Meaning on read	Action on write
0	Interrupt request disabled	No action, write is ignored
1	Interrupt request enabled	Disable interrupt request

For more information about counter overflow interrupt requests see *CP15 c9, Interrupt Enable Set Register, PMINTENSET* on page 2-62.

Accessing the PMINTENCLR Register

To access the PMINTENCLR Register, read or write the CP15 registers with <opc1> set to 0, <CRn> set to c9, <CRm> set to c14, and <opc2> set to 2. For example:

```
MRC p15,0,<Rt>,c9,c14,2    : Read Interrupt Enable Clear Register
MCR p15,0,<Rt>,c9,c14,2    : Write Interrupt Enable Clear Register
```

Appendix A
Recommended Memory-mapped and External Debug Interface for the Performance Monitors

This chapter describes the recommended memory-mapped and external debug interface to the performance monitors. It contains the following sections:

- *PMU memory-mapped register summary* on page A-68
- *PMU memory-mapped register descriptions* on page A-71.

A.1 PMU memory-mapped register summary

A memory-mapped interface enables software running on any processor in the system to access counters in the performance monitors. Table A-1 shows the memory-mapped registers for the performance monitors, in register offset order.

Note

An implementation must ensure that the 4KB region containing the PMU registers immediately follows, in physical memory, the 4KB region containing the debug registers.

All of the registers shown in Table A-1 are 33 bits wide.

Table A-1 PMU memory-mapped registers

Offset	Type	Name	Description
0x0*nn*	RW	PMXEVCNTR	*CP15 c9, Event Count Register, PMXEVCNTR* on page 2-60. *nn* is 4 times the counter number
0x07C	RW	PMCCNTR	*CP15 c9, Cycle Count Register, PMCCNTR* on page 2-53
0x080-0x3FC	-	-	Reserved, UNK/SBZP
0x4*nn*	RW	PMXEVTYPER	*CP15 c9, Event Type Select Register, PMXEVTYPER* on page 2-55. *nn* is 4 times the counter number.
0x47C	RW	PMCCFILTR	*CP15 c9, Cycle Count Filter Control Register, PMCCFILTR* on page 2-58
0x480-0xBFC	-	-	Reserved, UNK/SBZP
0xC00	RW	PMCNTENSET	*CP15 c9, Count Enable Set Register, PMCNTENSET* on page 2-44
0xC04-0xC1C	-	-	Reserved, UNK/SBZP
0xC20	RW	PMCNTENCLR	*CP15 c9, Count Enable Clear Register, PMCNTENCLR* on page 2-45
0xC24-0xC3C	-	-	Reserved, UNK/SBZP
0xC40	RW	PMINTENSET	*CP15 c9, Interrupt Enable Set Register, PMINTENSET* on page 2-62
0xC44-0xC5C	-	-	Reserved, UNK/SBZP
0xC60	RW	PMINTENCLR	*CP15 c9, Interrupt Enable Clear Register, PMINTENCLR* on page 2-64
0xC64-0xC7C	-	-	Reserved, UNK/SBZP
0xC80	RW	PMOVSR	*CP15 c9, Overflow Flag Status Register, PMOVSR* on page 2-47

Table A-1 PMU memory-mapped registers (continued)

Offset	Type	Name	Description
0xC84-0xC9C	-	-	Reserved, UNK/SBZP
0xCA0	WO	PMSWINC	*CP15 c9, Software Increment Register, PMSWINC* on page 2-48
0xCA4-0xDFC	-	-	Reserved, UNK/SBZP
0xE00	RW	PMCFGR	*Configuration Register, PMCFGR* on page A-71
0xE04	RW	PMCR	*CP15 c9, Performance Monitor Control Register, PMCR* on page 2-40
0xE08	RW	PMUSERENR	*CP15 c9, User Enable Register, PMUSERENR* on page 2-61
0xE0C-0xE1C	-	-	Reserved, UNK/SBZP
0xE20	RO	PMCEID0	*CP15 c9, Common Event Identification Registers 0 and 1, PMCEID0 and PMCEID1* on page 2-51
0xE24	RO	PMCEID1	*CP15 c9, Common Event Identification Registers 0 and 1, PMCEID0 and PMCEID1* on page 2-51
0xE28-0xE7C	-	-	Reserved, UNK/SBZP
0xE80-0xEFC	-	-	Reserved for IMPLEMENTATION DEFINED integration registers, UNK/SBZP
0xF00	-	-	Reserved for integration mode control register, RAZ/WI
0xF04-0xF9C	-	-	Reserved, UNK/SBZP
0xFA0	-	PMCLAIMSET	Reserved for Claim Set Register, RAZ/WI
0xFA4	-	PMCLAIMCLR	Reserved for Claim Tag Clear Register, RAZ/WI
0xFA8-0xFAC	-	-	Reserved, UNK/SBZP
0xFB0	WO	PMLAR	*Lock Access Register, PMLAR* on page A-72
0xFB4	RO	PMLSR	*Lock Status Register, PMLSR* on page A-73
0xFB8	RO	PMAUTHSTATUS	*Authentication Status Register, PMAUTHSTATUS* on page A-74
0xFBC-0xFC4	-	-	Reserved
0xFC8	-	PMDEVID	Reserved for DEVID Register, RAZ/WI
0xFCC	R	PMDEVTYPE	*Device Type Register, PMDEVTYPE* on page A-76

Table A-1 PMU memory-mapped registers (continued)

Offset	Type	Name	Description
0xFD0	RO	PMPID4	*Performance Monitor Peripheral Identification Register 4, PMPID4* on page A-79
0xFD4-0xFDC	-	PMPID5-PMPID7	Reserved for Performance Monitor Peripheral Identification Registers 5-7
0xFE0	RO	PMPID0	*Performance Monitor Peripheral Identification Register 0, PMPID0* on page A-77
0xFE4	RO	PMPID1	*Performance Monitor Peripheral Identification Register 1, PMPID1* on page A-77
0xFE8	RO	PMPID2	*Performance Monitor Peripheral Identification Register 2, PMPID2* on page A-78
0xFEC	RO	PMPID3	*Performance Monitor Peripheral Identification Register 3, PMPID3* on page A-78
0xFF0	RO	PMCID0	*Performance Monitor Component Identification Register 0, PMCID0* on page A-79
0xFF4	RO	PMCID1	*Performance Monitor Component Identification Register 1, PMCID1* on page A-80
0xFF8	RO	PMCID2	*Performance Monitor Component Identification Register 2, PMCID2* on page A-80
0xFFC	RO	PMCID3	*Performance Monitor Component Identification Register 3, PMCID3* on page A-81

A.2 PMU memory-mapped register descriptions

This section describes the memory-mapped registers for the performance monitors. Table A-1 on page A-68 provides cross references to individual registers.

A.2.1 Configuration Register, PMCFGR

The PMCFGR Register characteristics are:

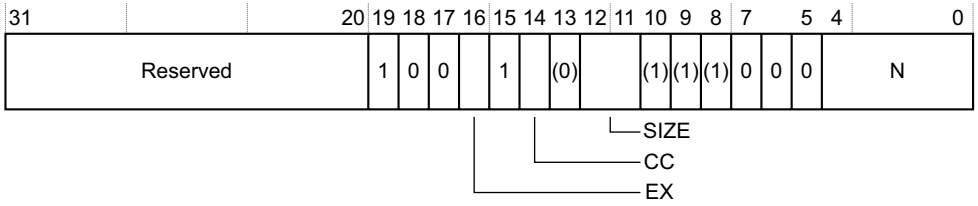
Purpose Contains PMU-specific configuration data.

Usage constraints There are no usage constraints.

Configurations Optional. Implemented only as part of the recommended performance monitors.

Attributes See the register summary in Table A-1 on page A-68. The value of this register is `0x0009DF`*nn*, where *nn* is the number of implemented counters.

The PMCFGR Register bit assignments are:



Bits[31:20] Reserved, UNK/SBZP.

Bit[19] Reserved, RAO.

Bit[18:17] Reserved, RAZ.

EX, bit[16] Export supported. This bit is RAO:

1 Export is supported. PMCR.X is writable.

Bit[15] Reserved, RAO.

CC, bit[14] Cycle counter implemented. This bit is RAO:

1 Cycle counter implemented. PMCR.C is writable.

Bit[13] Reserved, UNK/SBZP.

SIZE, bits[12:11]

Counter size. This is a read-only field that reads as 0b11.

0b11 32-bit counters.

Bits[10:8] Reserved, UNK/SBOP.

Bits[7:5] Reserved, RAZ.

N, bits[4:0] Number of event counters. This is a read-only field, with a value that indicates the number of implemented event counters, from 0b00000 if the implementation has no event counters, to 0b11111 if it has 31 event counters.

Note

The cycle counter, if implemented, is not included in the value indicated by the N field.

A.2.2 Lock Access Register, PMLAR

The PMLAR characteristics are:

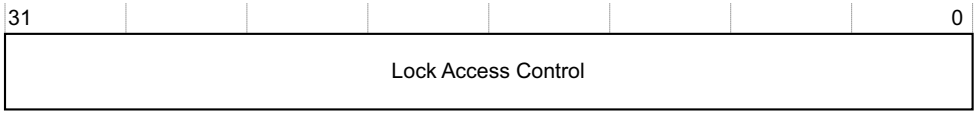
Purpose Provides a software lock on writes to the performance monitor registers through the memory-mapped interface. Use of this lock mechanism reduces the risk of accidental damage to the contents of the performance monitor registers. It does not prevent all accidental or malicious damage.

Usage constraints Used in conjunction with the PMLSR, see *Lock Status Register, PMLSR* on page A-73.The Lock Access Register is UNPREDICTABLE in the external debug interface.

Configurations Optional. Implemented only as part of the recommended performance monitors. If implemented in a processor that includes the Security Extensions, this is a Common register.

Attributes See the register summary in Table A-1 on page A-68.

The PMLAR bit assignments are:



Lock Access control, bits[31:0]

Writing the key value `0xC5ACCE55` to this field clears the lock, enabling write accesses to the performance monitor registers through the memory-mapped interface.

Writing any other value to this register sets the lock, disabling write accesses to the performance monitor registers through the memory-mapped interface.

A debugger checks the status of the software lock by reading the PMLSR, see *Lock Status Register, PMLSR* on page A-73.

Note

- The software lock of the performance monitor registers is independent of the software lock of the debug registers.
- The PMLAR is not required to be maintained over a power-down.

A.2.3 Lock Status Register, PMLSR

The PMLSR characteristics are:

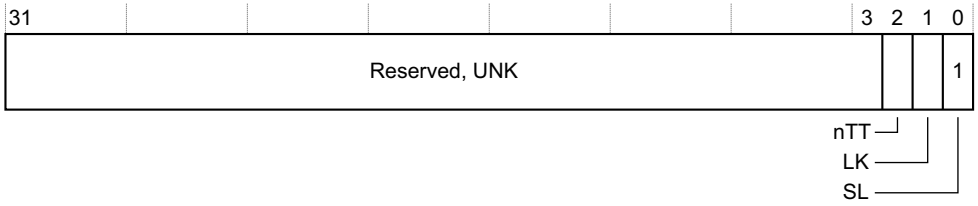
Purpose Shows the status of the software lock on the performance monitor registers.

Usage constraints Used in conjunction with the PMLAR, see *Lock Access Register, PMLAR* on page A-72. The Lock Status Register is RAZ in the external debug interface.

Configurations Optional. Implemented only as part of the recommended performance monitors. If implemented in a processor that includes the Security Extensions, this is a Common register.

Attributes See the register summary in Table A-1 on page A-68.

The PMLSR bit assignments are:



Bits[31:3] Reserved, UNK.

nTT, bit[2] Not 32-bit. This bit is always RAZ. It indicates that software must perform a 32-bit access to write the key to the Lock Access Register.

LK, bit[1] Locked bit. This bit indicates the status of the performance monitor registers lock. The possible values are:

0 Lock clear.

1 Lock set.

The reset value of this bit is 1.

SL, bit[0] Lock implemented. This bit is RAO.

A debugger sets or clears the performance monitor registers lock by writing to the PMLAR, see *Lock Access Register, PMLAR* on page A-72.

Note

- The software lock of the performance monitor registers is independent of the software lock of the debug registers.
- The PMLSR is not required to be maintained over a power-down.

A.2.4 Authentication Status Register, PMAUTHSTATUS

The PMAUTHSTATUS Register characteristics are:

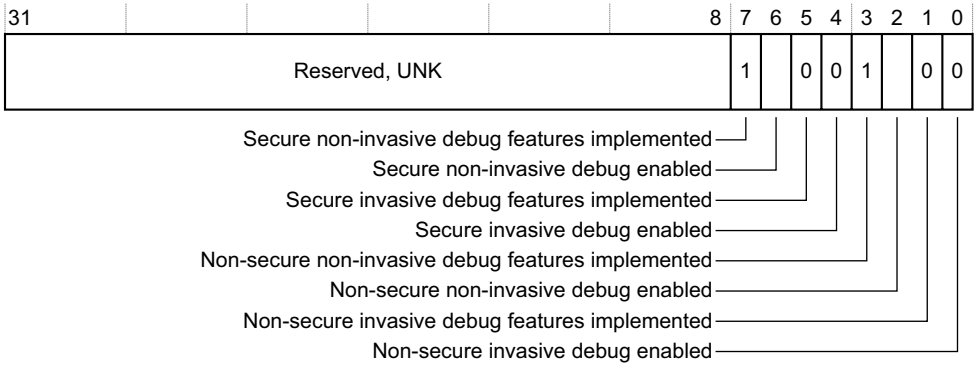
Purpose Indicates the implemented debug features and provides the current values of the configuration inputs that determine the debug permissions.

Usage constraints There are no usage constraints.

Configurations Optional. Implemented only as part of the recommended performance monitors. If implemented in a processor that includes the Security Extensions, this is a Common register.

Attributes See the register summary in Table A-1 on page A-68.

When the Security Extensions are implemented, the PMAUTHSTATUS Register bit assignments are:



Bits[31:8] Reserved, UNK.

Secure non-invasive debug features implemented, bit[7]

This bit is RAO, Secure non-invasive debug features are implemented.

Secure non-invasive debug enabled, bit[6]

This bit indicates whether non-invasive debug is permitted in Secure privileged modes. For the recommended authentication signal interface, this bit is the logical result of (**DBGEN** OR **NIDEN**) AND (**SPIDEN** OR **SPNIDEN**).

Secure invasive debug features implemented, bit[5]

This bit is RAZ, Secure invasive debug features are not implemented.

Secure invasive debug enabled, bit[4]

This bit is RAZ.

Non-secure non-invasive debug features implemented, bit[3]

This bit is RAO, Non-secure non-invasive debug features are implemented.

Non-secure non-invasive debug enabled, bit[2]

This bit indicates whether non-invasive debug is enabled. For the recommended authentication signal interface, this bit indicates the logical result of (**DBGEN** OR **NIDEN**).

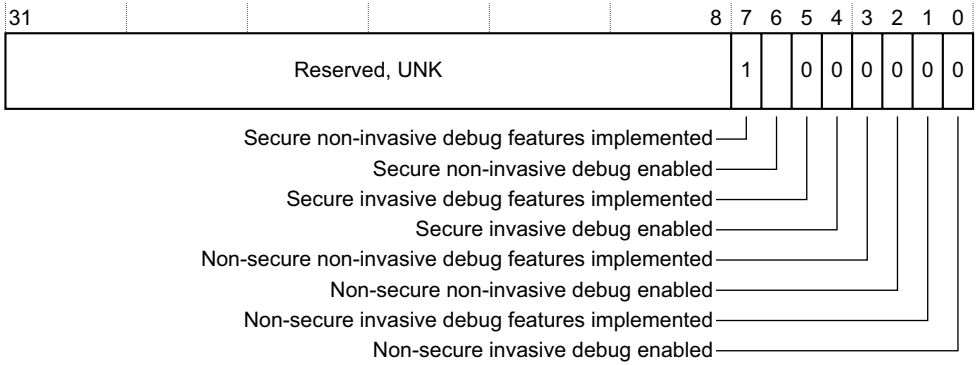
Non-secure invasive debug features implemented, bit[1]

This bit is RAZ, Non-secure invasive debug features are not implemented.

Non-secure invasive debug enabled, bit[0]

This bit is RAZ. It indicates whether invasive debug is enabled.

When the Security Extensions are not implemented, the PMAUTHSTATUS Register bit assignments are:



Bits[31:8] Reserved, UNK.

Secure non-invasive debug features implemented, bit[7]

This bit is RAO, Secure non-invasive debug features are implemented.

Secure non-invasive debug enabled, bit[6]

This bit indicates whether non-invasive debug is permitted in Secure privileged modes. For the recommended authentication signal interface, this bit is the logical result of (**DBGEN** OR **NIDEN**).

Secure invasive debug features implemented, bit[5]

This bit is RAZ, Secure invasive debug features are not implemented.

Secure invasive debug enabled, bit[4]

This bit is RAZ. It indicates whether invasive halting debug is permitted in Secure privileged modes.

Non-secure non-invasive debug features implemented, bit[3]

This bit is RAZ, Non-secure non-invasive debug features are not implemented.

Non-secure non-invasive debug enabled, bit[2]

This bit is RAZ. It indicates whether non-invasive debug is enabled.

Non-secure invasive debug features implemented, bit[1]

This bit is RAZ, Non-secure invasive debug features are not implemented.

Non-secure invasive debug enabled, bit[0]

This bit is RAZ. It indicates whether invasive debug is enabled.

A.2.5 Device Type Register, PMDEVTYPE

The PMDEVTYPE Register characteristics are:

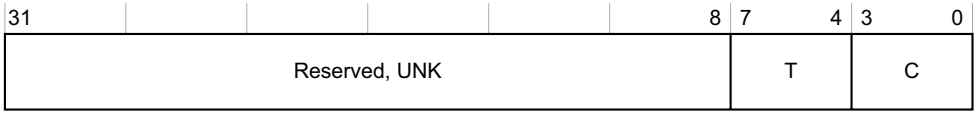
Purpose Provides the CoreSight device type information for the performance monitors. Every CoreSight component must implement the Device Type Register, that indicates the type of debug component.

Usage constraints There are no usage constraints.

Configurations Optional. Implemented only as part of the recommended performance monitors. If implemented in a processor that includes the Security Extensions, this is a Common register.

Attributes See the register summary in Table A-1 on page A-68.

The PMDEVTYPE Register bit assignments are:



Bits[31:8] Reserved, UNK.

T, bits[7:4] Sub type bit. This field reads as `0x1`, indicating a processor.

C, bits[3:0] Main class bit. This field reads as `0x6`, indicating performance monitors.

For more information about the CoreSight registers, see the *CoreSight Architecture Specification*.

A.2.6 Performance Monitor Peripheral Identification Register 0, PMPID0

The PMPID0 Register characteristics are:

Purpose Provides bits[7:0] of the 64-bit conceptual Peripheral ID.

Usage constraints There are no usage constraints.

Configurations Optional. Implemented only as part of the recommended performance monitors. If implemented in a processor that includes the Security Extensions, this is a Common register.

Attributes See the register summary in Table A-1 on page A-68.

The PMPID0 Register bit assignments are:

Bits[31:8] Reserved, UNK.

Part number[7:0], bits[7:0]

Bits[7:0] of the IMPLEMENTATION DEFINED part number.

Note

This is the part number for the Performance monitors block. It is not the same part number as the processor Debug block.

A.2.7 Performance Monitor Peripheral Identification Register 1, PMPID1

The PMPID1 Register characteristics are:

Purpose Provides bits[15:8] of the 64-bit conceptual Peripheral ID.

Usage constraints There are no usage constraints.

Configurations Optional. Implemented only as part of the recommended performance monitors. If implemented in a processor that includes the Security Extensions, this is a Common register.

Attributes See the register summary in Table A-1 on page A-68.

The PMPID1 Register bit assignments are:

Bits[31:8] Reserved, UNK.

JEP Identity Code[3:0], bits[7:4]

Bits[3:0] of the IMPLEMENTATION DEFINED JEP Identity Code.

For a device designed by ARM the JEP106 Identity Code is `0x3B` and therefore this field is `0xB`.

Part number[11:8], bits[3:0]

Bits[11:8] of the IMPLEMENTATION DEFINED Part number. For more information see the Note in the Part number field in *Performance Monitor Peripheral Identification Register 0, PMPID0* on page A-77.

A.2.8 Performance Monitor Peripheral Identification Register 2, PMPID2

The PMPID2 Register characteristics are:

Purpose Provides bits[23:16] of the 64-bit conceptual Peripheral ID.

Usage constraints There are no usage constraints.

Configurations Optional. Implemented only as part of the recommended performance monitors. If implemented in a processor that includes the Security Extensions, this is a Common register.

Attributes See the register summary in Table A-1 on page A-68.

The PMPID2 Register bit assignments are:

Bits[31:8] Reserved, UNK.

Revision, bits[7:4]

The IMPLEMENTATION DEFINED revision number for the device.

Uses JEP Code, bit[3]

For an ARMv7 implementation this bit must be one, indicating that the Peripheral ID uses a JEP106 Identity Code.

JEP Identity Code[6:4], bits[2:0]

Bits[6:4] of the IMPLEMENTATION DEFINED JEP Identity Code.

For a device designed by ARM the JEP106 Identity Code is `0x3B` and therefore this field is 0b011.

A.2.9 Performance Monitor Peripheral Identification Register 3, PMPID3

The PMPID3 Register characteristics are:

Purpose Provides bits[31:24] of the 64-bit conceptual Peripheral ID.

Usage constraints There are no usage constraints.

Configurations Optional. Implemented only as part of the recommended performance monitors. If implemented in a processor that includes the Security Extensions, this is a Common register.

Attributes See the register summary in Table A-1 on page A-68.

The PMPID3 Register bit assignments are:

Bits[31:8] Reserved, UNK.

RevAnd, bits[7:4]

The IMPLEMENTATION DEFINED manufacturing revision number for the device.

Customer modified, bits[3:0]

An IMPLEMENTATION DEFINED value that indicates an endorsed modification to the device.

If the system designer cannot modify the RTL supplied by the processor designer then this field is RAZ.

A.2.10 Performance Monitor Peripheral Identification Register 4, PMPID4

The PMPID4 Register characteristics are:

Purpose Provides bits[39:32] of the 64-bit conceptual Peripheral ID.

Usage constraints There are no usage constraints.

Configurations Optional. Implemented only as part of the recommended performance monitors. If implemented in a processor that includes the Security Extensions, this is a Common register.

Attributes See the register summary in Table A-1 on page A-68.

The PMPID4 Register bit assignments are:

Bits[31:8] Reserved, UNK.

4KB count, bits[7:4]

This field is RAZ for all ARMv7 implementations.

JEP106 Continuation code, bits[3:0]

The IMPLEMENTATION DEFINED JEP106 Continuation code.

For a device designed by ARM this field is `0x4`.

A.2.11 Performance Monitor Component Identification Register 0, PMCID0

The PMCID0 Register characteristics are:

Purpose Provides bits[7:0] of the 32-bit conceptual Component ID.

Usage constraints There are no usage constraints.

Configurations Optional. Implemented only as part of the recommended performance monitors. If implemented in a processor that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table A-1 on page A-68.

The PMCID0 Register bit assignments are:

Bits[31:8] Reserved, UNK.

Preamble byte 0, bits[7:0]

This byte has the value `0x0D`.

A.2.12 Performance Monitor Component Identification Register 1, PMCID1

The PMCID1 Register characteristics are:

Purpose Provides bits[15:8] of the 32-bit conceptual Component ID.

Usage constraints There are no usage constraints.

Configurations Optional. Implemented only as part of the recommended performance monitors. If implemented in a processor that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table A-1 on page A-68.

The PMCID1 Register bit assignments are:

Bits[31:8] Reserved, UNK.

Component class, bits[7:4]

This field has the value `0x9`, indicating an ARM Debug component.

Preamble, bits[3:0]

This field has the value `0x0`.

A.2.13 Performance Monitor Component Identification Register 2, PMCID2

The PMCID2 Register characteristics are:

Purpose Provides bits[23:16] of the 32-bit conceptual Component ID.

Usage constraints There are no usage constraints.

Configurations Optional. Implemented only as part of the recommended performance monitors. If implemented in a processor that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table A-1 on page A-68.

The PMCID2 Register bit assignments are:

Bits[31:8] Reserved, UNK.

Preamble byte 2, bits[7:0]

This field has the value `0x05`.

A.2.14 Performance Monitor Component Identification Register 3, PMCID3

The PMCID3 Register characteristics are:

Purpose Provides bits[31:24] of the 32-bit conceptual Component ID.

Usage constraints There are no usage constraints.

Configurations Optional. Implemented only as part of the recommended performance monitors. If implemented in a processor that includes the Security Extensions, this is a Common register.

Attributes See the registers summary in Table A-1 on page A-68.

The PMCID3 Register bit assignments are:

Bits[31:8] Reserved, UNK.

Preamble byte 3, bits[7:0]

This field has the value `0xB1`.

Glossary

Abort Is caused by an illegal memory access. Aborts can be caused by the external memory system or the MMU or MPU.

Addressing mode
Means a method for generating the memory address used by a load/store instruction.

Advanced SIMD
Is an extension to the ARM architecture that provides SIMD operations on a bank of extension registers. If the VFP extension is also implemented, the two extensions share the register bank and the SIMD operations include single-precision floating-point SIMD operations.

Aligned Refers to data items stored in such a way that their address is divisible by the highest power of 2 that divides their size. Aligned halfwords, words and doublewords therefore have addresses that are divisible by 2, 4 and 8 respectively.

An aligned access is one where the address of the access is aligned to the size of an element of the access

ARM instruction
Is a word that specifies an operation for a processor in ARM state to perform. ARM instructions must be word-aligned.

Byte Is an 8-bit data item.

Cache line
Is the basic unit of storage in a cache. Its size is always a power of two (usually 4 or 8 words), and must be aligned to a suitable memory boundary. A *memory cache line* is a block of memory locations with the same size and alignment as a cache line. Memory cache lines are sometimes loosely just called cache lines.

Conditional execution
Means that if the condition code flags indicate that the corresponding condition is true when the instruction starts executing, it executes normally. Otherwise, the instruction does nothing.

Exception
Handles an event. For example, an exception could handle an external interrupt or an Undefined Instruction.

Exception modes
Are privileged modes that are entered when specific exceptions occur.

Fault Is an abort that is generated by the MMU.

IMPLEMENTATION DEFINED
Means that the behavior is not architecturally defined, but should be defined and documented by individual implementations.

Long branch
Is the use of a load instruction to branch to anywhere in the 4GB address space.

Memory Management Unit (MMU)
Provides detailed control of a memory system. Most of the control is provided via translation tables held in memory.

Memory Protection Unit (MPU)
Is a hardware unit whose registers provide simple control of a limited number of protection regions in memory.

MMU *See* Memory Management Unit.

MPU *See* Memory Protection Unit.

Page table walk
See Translation table walk.

Prefetching
Is the process of fetching instructions from memory before the instructions that precede them have finished executing. Prefetching an instruction does not mean that the instruction has to be executed.

Privileged mode
Is any processor mode other than User mode. Memory systems typically check memory accesses from privileged modes against supervisor access permissions rather than the more restrictive user access permissions. The use of some instructions is also restricted to privileged modes.

RAO *See* Read-As-One.

RAZ *See* Read-As-Zero.

RAO/SBOP
Read-As-One, Should-Be-One-or-Preserved on writes.

In any implementation, the bit must read as 1, or all 1s for a bit field, and writes to the field must be ignored.

Software can rely on the bit reading as 1, or all 1s for a bit field, but must use an SBOP policy to write to the field.

RAO/WI Read-As-One, Writes Ignored.

In any implementation, the bit must read as 1, or all 1s for a bit field, and writes to the field must be ignored

Software can rely on the bit reading as 1, or all 1s for a bit field, and on writes being ignored.

RAZ/SBZP
Read-As-Zero, Should-Be-Zero-or-Preserved on writes.

In any implementation, the bit must read as 0, or all 0s for a bit field, and writes to the field must be ignored.

Software can rely on the bit reading as 0, or all 0s for a bit field, but must use an SBZP policy to write to the field.

RAZ/WI Read-As-Zero, Writes Ignored.

In any implementation, the bit must read as 0, or all 0s for a bit field, and writes to the field must be ignored.

Software can rely on the bit reading as 0, or all 0s for a bit field, and on writes being ignored.

Read-As-One (RAO)
In any implementation, the bit must read as 1, or all 1s for a bit field.

Read-As-Zero (RAZ)
In any implementation, the bit must read as 0, or all 0s for a bit field.

Reserved
Unless otherwise stated:

- instructions that are reserved or that access reserved registers have UNPREDICTABLE behavior
- bit positions described as Reserved are UNK/SBZP.

SBO *See* Should-Be-One.

SBOP *See* Should-Be-One-or-Preserved.

SBZ *See* Should-Be-Zero.

SBZP *See* Should-Be-Zero-or-Preserved.

Should-Be-One (SBO)
Should be written as 1, or all 1s for a bit field, by software. Values other than 1 produce UNPREDICTABLE results.

Should-Be-One-or-Preserved (SBOP)
Must be written as 1, or all 1s for a bit field, by software if the value is being written without having been previously read, or if the register has not been initialized. Where the register was previously read on the same processor, since the processor was last reset, the value in the field should be preserved by writing the value that was previously read.

Hardware must ignore writes to these fields.

If a value is written to the field that is neither 1 (or all 1s for a bit field), nor a value previously read for the same field on the same processor, the result is UNPREDICTABLE.

Should-Be-Zero (SBZ)
Should be written as 0, or all 0s for a bit field, by software. Values other than 0 produce UNPREDICTABLE results.

Should-Be-Zero-or-Preserved (SBZP)
Must be written as 0, or all 0s for a bit field, by software if the value is being written without having been previously read, or if the register has not been initialized. Where the register was previously read on the same processor, since the processor was last reset, the value in the field should be preserved by writing the value that was previously read.

Hardware must ignore writes to these fields.

If a value is written to the field that is neither 0 (or all 0s for a bit field), nor a value previously read for the same field on the same processor, the result is UNPREDICTABLE.

SIMD Means Single-Instruction, Multiple-Data operations.

Simple sequential execution
The behavior of an implementation that fetches, decodes and completely executes each instruction before proceeding to the next instruction. Such an implementation performs no speculative accesses to memory, including to instruction memory. The implementation does not pipeline any phase of execution. In practice, this is the theoretical execution model that the architecture is based on, and ARM does not expect this model to correspond to a realistic implementation of the architecture.

Thumb instruction
Is one or two halfwords that specify an operation for a processor in Thumb state to perform. Thumb instructions must be halfword-aligned.

TLB *See* Translation Lookaside Buffer.

Translation Lookaside Buffer (TLB)
Is a memory structure containing the results of translation table walks. They help to reduce the average cost of a memory access. Usually, there is a TLB for each memory interface of the ARM implementation.

Translation tables
Are tables held in memory. They define the properties of memory areas of various sizes from 1KB to 1MB.

Translation table walk
Is the process of doing a full translation table lookup. It is performed automatically by hardware.

A translation table walk is also called a page table walk.

Unaligned
An unaligned access is an access where the address of the access is not aligned to the size of an element of the access.

Unaligned memory accesses
Are memory accesses that are not, or might not be, appropriately halfword-aligned, word-aligned, or doubleword-aligned.

UNDEFINED
Indicates an instruction that generates an Undefined Instruction exception.

Unified cache
Is a cache used for both processing instruction fetches and processing data loads and stores.

UNKNOWN
An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not be a security hole. UNKNOWN values must not be documented or promoted as having a defined value or effect.

UNK/SBOP
UNKNOWN on reads, Should-Be-One-or-Preserved on writes.

In any implementation, the bit must read as 1, or all 1s for a bit field, and writes to the field must be ignored.

Software must not rely on the bit reading as 1, or all 1s for a bit field, and must use an SBOP policy to write to the field.

UNK/SBZP
UNKNOWN on reads, Should-Be-Zero-or-Preserved on writes.

In any implementation, the bit must read as 0, or all 0s for a bit field, and writes to the field must be ignored.

Software must not rely on the bit reading as 0, or all 0s for a bit field, and must use an SBZP policy to write to the field.

UNK Is an abbreviation indicating that software must treat a field as containing an UNKNOWN value.

In any implementation, the bit must read as 0, or all 0s for a bit field. Software must not rely on the field reading as zero.

UNPREDICTABLE
Means the behavior cannot be relied upon. UNPREDICTABLE behavior must not represent security holes. UNPREDICTABLE behavior must not halt or hang the processor, or any parts of the system. UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

VFP Is a coprocessor extension to the ARM architecture. It provides single-precision and double-precision floating-point arithmetic.

Word Is a 32-bit data item. Words are normally word-aligned in ARM systems.

Word-aligned
Means that the address is divisible by 4.

Write-Allocate cache
Is a cache in which a cache miss on storing data causes a cache line to be allocated into the cache.

Write-Back cache
Is a cache in which when a cache hit occurs on a store access, the data is only written to the cache. Data in the cache can therefore be more up-to-date than data in main memory. Any such data is written back to main memory when the cache line is cleaned or re-allocated. Another common term for a Write-Back cache is a *copy-back cache*.

Write-Through cache

Is a cache in which when a cache hit occurs on a store access, the data is written both to the cache and to main memory. This is normally done via a write buffer, to avoid slowing down the processor.

Write buffer

Is a block of high-speed memory whose purpose is to optimize stores to main memory.